Systems

Corporate Strategy & Intelligence Dossier

Prepared for	Project-X Executive Leadership Team
Prepared by	Strategy & Compliance Office
Document date	October 23, 2025
Classification	Strictly Confidential – Do Not Distribute

Table of Contents

User Management System

Location: /server/src/modules/auth

TL;DR

The user management system anchors identity, authentication, and session governance for the AI Governance Platform.

It lives in server/src/modules/auth, coordinating JWT issuance, Casbin enforcement, and user lifecycle workflows.

This guide walks through the service structure, login and recovery flows, administrative tooling, and the client integrations that depend on it.

1. System Overview

The user management system is implemented in the backend Auth Service located at server/src/modules/auth. It coordinates user lifecycle operations, session control, and security integrations for the AI Governance Platform. The service relies on JWT-based authentication, Casbin RBAC enforcement, and Nodemailer to deliver transactional emails.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L74-L86■■F:docs/02-technical-specifications/06-security-implementation.md†L54-L75■

2. Module Structure

The auth module follows the platform's feature-based convention: controllers expose Express handlers, services encapsulate business logic, and repositories manage database access. It collaborates with shared middleware for token validation, utilities for hashing, integrations for email delivery, and the routes layer that mounts /auth endpoints.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L52-L86■

2.1 Registration

- **Endpoint:** POST /auth/register

- **Flow:**

1. Validate input payload (email, password, organization context).

2. Hash the password with bcrypt (≥12 rounds) and persist the user record with default RBAC role assignments.

3. Trigger optional verification or welcome emails via Nodemailer integration.

4. Log the event for audit purposes and apply rate limiting to prevent abuse.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L74-L86■■F:docs/02-technical-specifications/06-security-implementation.md†L54-L75■

2.2 Login

- **Endpoint:** POST /auth/login
- **Flow:**

1. Authenticate credentials against stored bcrypt hashes.

2. Issue a signed JWT containing user ID, role, and expiration, and generate a refresh token with longer validity for session continuity.

3. Store refresh token metadata (device, IP, expiry) for revocation and anomaly monitoring.

4. Enforce rate limiting and log attempts for security analytics.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L74-L86■■F:docs/02-technical-specifications/06-security-implementation.md†L54-L75■

2.3 Logout

- **Endpoint:** POST /auth/logout
- **Flow:**

1. Invalidate the active refresh token by removing or flagging the record in the session store.

2. Record the logout event in audit logs, including device and timestamp.

3. Optionally revoke other active sessions through administrative controls when suspicious activity is detected.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L74-L86■■F:docs/01-about/04-security-and-data-protection.md†L230-L249■

2.4 Refresh Tokens

- **Endpoint:** POST /auth/refresh
- **Flow:**

1. Validate the refresh token against stored metadata and enforce rotation policies.

2. Issue a new access token and (optionally) a new refresh token with updated expiry limits.

3. Reject reused or expired tokens and log invalidation attempts to support threat detection.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L74-L86■■F:docs/02-technical-specifications/06-security-implementation.md†L54-L75■

2.5 Password Reset Emails

- **Endpoints:** POST /auth/forgot-password, POST /auth/reset-password
- **Flow:**

1. Generate a time-bound reset token stored alongside user metadata.

2. Send password reset instructions via Nodemailer using templated content and secure links.

3. Validate the token, enforce password strength policies, and update the hashed password upon completion.

4. Log successful and failed reset attempts for compliance tracking.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L74-L80■

3. Session Handling and RBAC

- JWT validation middleware protects all authenticated routes and rehydrates user context on each request.
- Refresh tokens extend sessions without reauthentication but are constrained by timeout and rotation policies.
- Casbin-backed RBAC policies ensure users only access resources permitted to their role, with admin tooling to review and revoke sessions as needed.■F:docs/02-technical-specifications/06-security-implementation.md†L54-L95■■F:docs/01-about/04-security-and-data-protection.md†L206-L238■

4. Supported Authentication Methods

Security policies mandate multiple enterprise authentication choices that integrate with the auth module:

- **Single Sign-On (SSO):** Supports SAML 2.0 and OpenID Connect providers for federated login workflows.■F:docs/01-about/04-security-and-data-protection.md†L206-L216■
- **Multi-Factor Authentication (MFA):** Required for administrative and privileged accounts, with hooks in the login pipeline to validate second factors before issuing tokens.■F:docs/01-about/04-security-and-data-protection.md†L206-L216■

- **Passwordless Options:** Optional FIDO2 or hardware security keys can be registered to bypass passwords while maintaining strong assurance levels.■F:docs/01-about/04-security-and-data-protection.md†L206-L216■
- **Session Governance:** Automatic timeouts, refresh limits, and device tracking are enforced to manage risk across sessions and devices.■F:docs/01-about/04-security-and-data-protection.md†L206-L216■

5. Administrative Controls

Platform administrators operate dedicated tooling that interacts with the auth module to:

- Assign and delegate roles across organizational units while maintaining least-privilege access.■F:docs/01-about/04-security-and-data-protection.md†L200-L238■
- Configure SSO connections, enforce MFA policies, and manage passwordless enrollment options.■F:docs/01-about/04-security-and-data-protection.md†L206-L249■
- Revoke user access, expire active sessions, and orchestrate user lifecycle events with immutable logging for audits.■F:docs/01-about/04-security-and-data-protection.md†L230-L249■
- Execute break-glass procedures that grant temporary elevated access with real-time alerts and mandatory post-event reviews.■F:docs/01-about/04-security-and-data-protection.md†L253-L259■

6. Frontend Flows

The React client mirrors backend capabilities through dedicated auth views and context managers:

- client/src/features/auth/ hosts login, registration, password reset, and MFA setup pages aligned with the /auth API contract.■F:docs/02-technical-specifications/03-frontend-architecture.md†L50-L103■
- Global Auth Context maintains JWT tokens, refresh logic, and role metadata, ensuring protected routes enforce session state across layouts.■F:docs/02-technical-specifications/03-frontend-architecture.md†L96-L139■
- Security measures in the frontend (CSP, CSRF headers, session timeout handling) complement backend controls for a cohesive user experience during authentication.■F:docs/02-technical-specifications/03-frontend-architecture.md†L143-L160■

7. Related Documentation

- Backend Architecture & APIs – Auth Service overview.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L74-L86■
- Security Implementation – Authentication, session handling, and RBAC policies.■F:docs/02-technical-specifications/06-security-implementation.md†L54-L95■

- Security and Data Protection – Enterprise authentication methods and administrative governance.■F:docs/01-about/04-security-and-data-protection.md†L206-L259■

RBAC System

Location: /server/src/modules/auth

TL;DR

The AI Governance Platform enforces role-based access control (RBAC) using **Casbin** across the Node.js backend. Policies are persisted in **PostgreSQL**, cached for high throughput, and evaluated in middleware before any module logic runs. This document explains the role hierarchy, permission granularity, enforcement layers, and operational workflows that keep access secure and auditable.

---

1. System Overview

The RBAC system resides primarily in server/src/modules/auth, where controllers orchestrate login, token issuance, and authorization decisions. A dedicated Casbin adapter loads policies from PostgreSQL into an in-memory enforcer instance that is shared across Express middleware.

Key characteristics:

- **Policy Model:** rbac_with_domains_model.conf implements subject ■ domain ■ object ■ action rules, enabling tenant-specific scoping.
- **Policy Persistence:** The Casbin adapter writes to and reads from the auth_policies table via Prisma. Batched transactions ensure atomic role updates.
- **Enforcement Surface:** Middleware in server/src/middleware/authorization.js checks each request before handing control to feature controllers (Auth, Governance Engine, Frameworks, Evidence, Notifications, Tasks).
- **Observability:** Authorization decisions are logged to audit_logs with request metadata to satisfy compliance requirements and support forensic analysis.

2. Role Hierarchy

RBAC follows a hierarchical inheritance model. Each role inherits the permissions of the roles beneath it while adding scoped abilities aligned with least-privilege principles.

2.1 Built-in Roles

Role	*Inherits*	*Core Capabilities*
Super Admin	Admin	Platform configuration, tenant provisioning, policy bootstrap.
Admin	Compliance Officer	Role management, policy authoring, integration secrets, manual overrides.
Compliance Officer	Auditor	Framework CRUD, control mapping, evidence approval, access reviews.
Auditor	Engineer	Read-only dashboards, report exports, audit log retrieval.
Engineer	System Service	Evidence submission, probe configuration, remediation task updates.
System Service	*None*	Machine-to-machine automation (ingest controls, run scheduled jobs).

Role inheritance is expressed through Casbin g and g2 relationships and synchronized automatically when seeding or updating policies.

2.2 Custom Roles

- Tenants can define custom roles that derive from any built-in role or another custom definition.
- Custom roles are stored in the auth_roles table with metadata about owner tenant, description, and review cadence.
- Administrators assign capabilities by mapping custom roles to granular policies (e.g., data steward with read/write access to AI datasets but no policy authoring rights).
- Casbin domains ensure that custom roles remain isolated per tenant while still benefiting from shared policy templates.

2.3 Separation-of-Duty Rules

To prevent conflicts of interest:

- Approval actions (e.g., evidence sign-off, framework publication) require a role different from the one that initiated the change. Casbin policies enforce this via `rule_effect = deny` conditions when subject and initiator match.
- Super Admins can impersonate users for troubleshooting, but impersonation tokens are logged and require post-event approval by a second Admin.
- Automated checks validate that no user simultaneously holds `Admin` and `Auditor` roles within the same tenant unless explicitly whitelisted for break-glass scenarios.

3. Permission Granularity

Permissions are defined using a combination of resource type, resource identifier, and action verb. Examples include:

- `framework:123:update` – modify metadata for framework `123`.
- `control:*:approve` – approve any control evidence submission.
- `governance-engine:run-evaluation` – trigger recalculation of risk scores.
- `user:tenant-456:assign-role` – manage user roles within tenant `456`.

Casbin policies support:

- **Row-level constraints** via domains (tenant, framework, control).
- **Action wildcards** for read-only vs. read-write bundles.
- **Conditional ABAC checks** (e.g., verifying that a user is assigned to the same control as the request) implemented through middleware context resolvers.

4. Enforcement Layers

4.1 Request Lifecycle

1. **Authentication Middleware** validates the JWT issued by the Auth Service and attaches `userId`, `roleIds`, and tenant metadata to `req.authContext`.

2. **Casbin Enforcement Middleware** (`requirePermission` helper) builds a subject string (`user` or `role`), domain (tenant/framework), object, and action from the request.

3. Casbin evaluates the tuple; on deny, it returns HTTP 403 with an audit log entry.

4. On allow, request handlers in the target module execute business logic, optionally making **secondary checks** (e.g., verifying framework status or workflow stage).

4.2 Module-Specific Enforcement

- **Auth Module (server/src/modules/auth)** – Houses Casbin adapter, role management APIs, and onboarding workflows. Controllers expose /roles, /policies, and /access-reviews endpoints that are themselves guarded by meta-policies to avoid privilege escalation.
- **Governance Engine (server/src/modules/governance)** – Uses helper utilities to ensure only authorized roles can evaluate controls, publish governance scores, or override outcomes. Sensitive operations such as forced remediation triggers require dual-authorization tokens generated by the Auth module.
- **Middleware Layer (server/src/middleware)** – Provides shared requireRole, requirePermission, and enforceSoD middleware functions reused across routers. Middleware caches policy decisions per request to minimize redundant Casbin calls.

5. Policy Storage and Caching

- **Database Tables:**
- auth_policies – Canonical Casbin policy rules (p, g, g2 records).
- auth_roles – Role metadata, inheritance trees, custom attributes.
- auth_policy_revisions – Append-only log tracking who changed a policy, when, and why.
- **Adapter:** server/src/modules/auth/casbin-adapter.js implements the Casbin adapter interface using Prisma for transactional reads/writes.
- **Caching:**
- Policy sets cached in Redis under casbin:tenant:<id> keys with a 5-minute TTL.
- Local in-process cache (LRU) avoids repeated adapter calls during burst traffic.
- **Invalidation:**
- Cache entries invalidated automatically on policy mutation events via a message bus (policy.updated topic) published from the Auth module.
- Manual invalidation available through /auth/policies/refresh (Admin-only) to handle cross-region synchronization.

6. Access Reviews and Governance Workflows

- **Quarterly Access Reviews:** Admins trigger campaigns that snapshot current role assignments, send review tasks to Compliance Officers, and require sign-off before completion.
- **Event-Driven Reviews:** Detect anomalies (e.g., dormant high-privilege accounts, overlapping Admin/Auditor assignments) and open remediation tasks in the Task Service.
- **Certification Evidence:** Completed reviews export certified reports stored in the Evidence Repository for audit readiness.
- **Delegation:** Temporary delegations create time-bound policies with automatic expiry enforced by scheduled jobs.

7. Updating Casbin Policies

1. **Plan the Change:** Document desired permission updates, referencing the guidance for API endpoints and controllers involved.

2. **Edit Policies:**

- Use the Admin UI or call the /auth/policies API to add/update p (permission) or g (role inheritance) rules.
- For bulk updates, utilize the RBAC sync CLI from the developer tooling repository to read YAML policy manifests and push them through the Casbin adapter.

3. **Validate:** Run the /auth/policies/dry-run endpoint to preview the effect on target subjects and ensure separation-of-duty constraints remain intact.

4. **Deploy:** Commit accompanying migration scripts if new roles or resources are introduced. Coordinate with DevOps to schedule cache invalidation.

5. **Audit:** Confirm that a new entry appears in auth_policy_revisions with justification details and reviewer sign-off.

8. Testing RBAC Scenarios

- **Unit Tests:**
- Located in server/src/modules/auth/__tests__/authorization.spec.js to cover policy evaluation logic.
- Mock the Casbin enforcer to simulate allow/deny paths.
- **Integration Tests:**
- API-level tests under server/src/routes/__tests__ exercise middleware with real policies loaded from the PostgreSQL test database.
- Use fixtures to seed tenants, roles, and policies per scenario.

- **Manual Verification:**
- Employ the shared Postman collection published with the API workspace for exploratory testing.
- Run npm run test:rbac to execute the focused RBAC test suite before releases.
- **Governance Engine Alignment:**
- Cross-validate that Governance Engine workflows respect RBAC outcomes by running the evaluation regression suite described in the .

9. Related Documentation

Notification System

Location: /server/src/modules/notifications

TL;DR

The notification system orchestrates outbound communication across email and enterprise collaboration channels.

It is centered around the server/src/modules/notifications package, which exposes shared workflow utilities, template

management, and channel adapters.

This guide outlines the email delivery lifecycle, integrations with ServiceNow, Jira, and Slack, and patterns for

extending notification coverage and observability.

1. Purpose and Scope

The notification system ensures that domain events are communicated to internal and external stakeholders using the correct

channel, format, and timing. The server/src/modules/notifications package consolidates reusable utilities for triggering

notifications, rendering user-facing content, and recording delivery outcomes. This document serves as an onboarding guide

for contributors who need to operate, extend, or integrate the notification stack.

2. Architecture Overview

2.1 Core Packages

The notification module is organised around the following packages and services:

Package / Service	*Description*
server/src/modules/notifications/events	Defines domain events, event-to-notification mapping rules, and orchestration pipelines.
server/src/modules/notifications/templates	Provides template registries, loaders, and helpers for multi-language rendering.
server/src/modules/notifications/channels	Channel-specific providers for email, ServiceNow, Jira, Slack, SMS, and future adapters.
server/src/modules/notifications/logging	Structured logging, correlation IDs, and audit trail utilities.
server/src/modules/notifications/config	Central configuration including delivery policies, throttles, and integration credentials.
server/src/modules/notifications/testing	Harnesses and fixtures for regression testing and contract validation.

2.2 Workflow Phases

1. **Event ingestion** – domain events or scheduled jobs enqueue notification intents.

2. **Eligibility checks** – policy evaluators confirm that recipients and channels are permitted.

3. **Template selection** – content builders resolve template IDs, locales, and variant overrides.

4. **Payload rendering** – dynamic data is merged into templates and attachments are generated if required.

5. **Channel dispatch** – the channel adapter sends the notification and returns a delivery status.

6. **Post-processing** – results are persisted, metrics are emitted, and follow-up actions (retries or escalations) are queued.

3. Email Workflows

3.1 Triggering Events

- Event producers publish messages to the notification queue via the events package.
- Cron tasks in server/src/modules/notifications/schedulers initiate digest or reminder messages.

- Manual overrides can be triggered through the operations console using the REST endpoint POST /notifications/trigger.

3.2 Template Management

- Templates reside in templates/email/<locale>/<notification-type>.hbs and are versioned through Git.
- Layouts and partials are composed using Handlebars helpers defined in templates/helpers.ts.
- Every template must declare metadata (ID, locale, subject, required context keys) in templates/registry.json.
- The registry is validated on boot. Invalid or missing templates emit structured warnings and are excluded from dispatch.
- Preview endpoints (GET /notifications/templates/:id/preview) allow operations teams to verify rendering with sample data.

3.3 Personalisation and Rendering

- Context builders pull profile data from the user service, and merge it with event payloads.
- Sensitive fields (PII, secrets) must be redacted using the maskSensitive helper before they are injected into templates.
- Email subjects and preheaders support localisation by mapping to language-specific keys stored in i18n/messages.json.
- Attachments are generated asynchronously and stored in the object store; the email body contains signed download links.

3.4 Dispatching and Delivery

- SMTP delivery defaults to AWS SES but can be switched per environment using NOTIFICATIONS_EMAIL_PROVIDER.
- All outbound requests use the shared retry policy (3 attempts, exponential backoff starting at 15 seconds).
- Responses are normalised to DeliveryResult objects that include provider message IDs, timestamps, and status codes.
- Bulk sends route through the background worker (notifications-email-consumer) to prevent API rate limit breaches.

3.5 Logging and Audit Trails

- Each email dispatch logs a correlation ID that links back to the originating domain event.
- Success, retry, and failure outcomes are emitted to the observability pipeline via notifications.logger.
- Delivery receipts are captured and stored in notifications_deliveries with the raw provider payload for later auditing.
- Logs older than 180 days are archived to cold storage in compliance with governance policies.

4. Channel Integrations

4.1 ServiceNow

Authentication

- Supports OAuth 2.0 client credentials with scopes limited to incident and task APIs.
- Credentials are stored in the secrets manager under servicenow/notifications and rotated every 90 days.

Event Triggers

- Incidents created or updated with priority ≥ P2 emit IncidentEscalated events mapped to ServiceNow notifications.
- Change management workflows trigger updates when approvals are due or overdue.

Synchronization Behaviour

- Delivery results update the originating ServiceNow incident via the /api/now/table/incident endpoint with a comment and

custom field indicating the notification status.

- A nightly reconciliation job confirms that all pending ServiceNow tasks have matching notification records.

4.2 Jira

Authentication

- Uses PAT (Personal Access Token) or OAuth 1.0a depending on deployment; tokens are scoped to project automation APIs.

- The token alias jira/notifications is loaded through the shared credentials provider at runtime.

Event Triggers

- Issue transitions to statuses such as Blocked, Ready for Review, or Done create notifications for assigned users.
- Sprint events (start/end) broadcast digest updates to watchers using batched email and Slack notifications.

Synchronization Behaviour

- Notification delivery updates Jira issues by posting to the /rest/api/3/issue/{key}/comment endpoint.
- Failed deliveries add a label (notification_failed) so project leads can audit follow-up actions.
- State changes triggered through notification acknowledgements (e.g., approve/reject links) are pushed back to Jira via webhooks.

4.3 Slack

Authentication

- Uses OAuth 2.0 bot tokens stored under slack/bot-token in the secrets manager.
- Tokens require the scopes chat:write, im:write, users:read, and channels:read.

Event Triggers

- Real-time events (critical incidents, policy approvals) push to dedicated channels configured in slack.channels.json.
- Daily digests combine lower-priority updates and are scheduled through the digest worker.

Synchronization Behaviour

- Delivery statuses are captured using Slack's chat.postMessage response metadata and stored for analytics.
- Message timestamps are saved so follow-up edits or deletions can be issued when notifications are acknowledged elsewhere.
- Slash commands (/notify-ack, /notify-snooze) feed back into the notification API, updating the master delivery record.

5. Extending Notification Types

5.1 Adding a New Notification Schema

1. Define a domain event payload in events/schemas with strict typing and validation rules.

2. Register the schema in events/index.ts, mapping it to the channels and templates that should respond.

3. Update the notification type enum (NotificationType) to include the new identifier and add documentation comments.

5.2 Implementing Channel Strategies

1. Extend channels/<channel>/strategies.ts with a handler implementing the shared ChannelStrategy interface.

2. Map the new notification type to the handler in the channel registry.

3. Provide any channel-specific configuration (webhook URLs, queue names) in config/channels.yml.

5.3 Registering Templates and Localisations

1. Create email templates and optional Slack/Jira message templates with consistent placeholder names.

2. Update templates/registry.json to register each locale-specific variant.

3. Add translation strings to i18n/messages.json and run yarn notifications:validate to ensure integrity.

6. Delivery Policies and Configuration

6.1 Scheduling and Throttling

- Scheduling rules live in config/delivery-policies.yml and support cron expressions, quiet hours, and blackout windows.
- The throttling engine enforces per-user and per-channel rate limits. Defaults: max 5 critical notifications per hour per user,

max 20 informational notifications per day per user.

- Bulk operations (imports, migrations) can request temporary overrides via the DeliveryPolicyOverride API.

6.2 Retries and Escalations

- Retry policies can be tuned per notification type with settings for attempt count, delay strategy, and jitter.
- Escalation rules promote unresolved incidents to alternative channels (e.g., from email to Slack DM) after configurable

timeouts.

- When retries exhaust, the system opens an incident in ServiceNow tagged with notification-delivery-failure.

6.3 User Preferences and Compliance

- User-level preferences stored in the profile service dictate opt-in/out status and preferred channels.
- GDPR compliance requires double opt-in for marketing-style notifications and explicit consent records per locale.
- Audit logs capture changes to delivery preferences and can be queried through the governance portal.

7. Monitoring and Failure Handling

7.1 Metrics and Dashboards

- Core metrics (queued, sent, delivered, failed, latency) are exported to Prometheus with the notifications_* prefix.
- Grafana dashboards visualise channel performance, top failing templates, and backlog depth.
- Custom dimensions (notification type, environment, provider) enable drill-down during incident response.

7.2 Alerting and Runbooks

- Alert rules trigger when failure rates exceed thresholds or when queue depth stays above 5,000 for more than 15 minutes.
- On-call rotations receive PagerDuty alerts with deep links to relevant Grafana dashboards and runbook pages.
- Runbooks outline triage steps: validate provider status, review recent deployments, replay failed messages, and file RCA.

7.3 Log Retention and Privacy

- Structured logs stream to the central log platform with retention of 30 days hot, 180 days warm, 365 days archived.
- PII redaction policies apply to logs, ensuring sensitive content is masked before leaving the application boundary.
- Access to archived logs requires break-glass approval and is monitored via security information and event management (SIEM).

Audit Logging and Monitoring

Location: /server/src/lib/logging

TL;DR

The audit logging and monitoring system guarantees immutable, end-to-end visibility into security-critical activity.

Built around server/src/lib/logging/audit.ts, it captures structured events, enforces retention and tamper controls, and feeds observability pipelines for governance oversight.

Use this guide to understand capture scope, storage contracts, operational safeguards, and developer instrumentation practices.

1. Log Capture Scope

- Capture all authentication events (login, logout, MFA challenges, failures) across user-facing and service-to-service flows.
- Record privileged actions, configuration changes, data exports, and any alteration of access controls in administrative interfaces.
- Instrument critical business operations, including financial transactions, data mutations, workflow state transitions, and API calls flagged as high risk.
- Track system-level events such as deployment lifecycle actions, infrastructure scaling, and security control updates.
- Ensure each audit entry includes timestamp (UTC), actor identity, action, target resource, request origin, and success or failure context.

2. Storage Targets

- **Application Layer:** Use Winston transports for structured JSON logs, tagging audit events with category: "audit" and unique correlation IDs. Deliver to rolling file transport for local debugging with rotation configured (<24 h of retention) and to standard output for container orchestration capture.

- **Centralized Logging:** Stream Winston output to the centralized logging pipeline (e.g., Elasticsearch/OpenSearch, Loki, or Splunk) via HTTPS or sidecar agents. Ensure dedicated audit index/stream with role-based access separation from general application logs.

3. Retention Policies

- Maintain audit logs in centralized storage for a minimum of 400 days to satisfy regulatory requirements; extend per jurisdictional mandates.
- Archive immutable snapshots to cold storage (object store with write-once, read-many policies) every 30 days for a minimum of 7 years.
- Automatically purge local file transport logs after 24 hours and confirm centralized lifecycle policies enforce retention schedules.

4. Immutability Controls

- Enable append-only permissions on audit indices and restrict deletion capabilities to the security/compliance team with change control approval.
- Utilize object storage retention policies or S3 Glacier Vault Lock-style controls to enforce WORM on archived snapshots.
- Sign audit batches with cryptographic hashes and store hash manifests in a separate integrity ledger to detect tampering.

5. Monitoring and Alerting Integration

- Feed audit streams into the security information and event management (SIEM) stack for correlation with security alerts and anomaly detection.
- Configure alerting rules for high-risk audit events (privilege escalations, repeated access denials, failed administrative logins) with notification channels to on-call security and operations teams.
- Link dashboards in the observability platform to audit log metrics (event volumes, failure rates) and include runbooks for triaging anomalous patterns.

6. Access Restrictions and Review Processes

- Grant read-only access to audit indices to the security, compliance, and incident response teams via least-privilege roles; deny direct access to engineering by default.
- Require break-glass procedures with approval logging for temporary investigative access outside the security team.

- Schedule quarterly formal reviews of audit log health, retention adherence, and sample integrity checks. Document findings and remediation actions in the compliance tracker.
- Implement weekly automated reports summarizing critical audit events, reviewed by security leadership with sign-off recorded in governance tooling.

7. Developer Instrumentation Guidelines

1. **Identify Audit-Worthy Actions:** When adding new features, flag actions that affect security posture, user data, or compliance-sensitive operations. Consult the security team if uncertain.

2. **Use Shared Audit Helpers:** Instrument events via the auditLogger utility (see src/lib/logging/audit.ts) to standardize metadata, correlation IDs, and JSON schema.

3. **Mask Sensitive Data:** Never log secrets, full payment card numbers, passwords, or personal data beyond minimal identifiers. Use the redactFields option provided by the audit logger to mask sensitive attributes before emission.

4. **Correlate Requests:** Attach request IDs, user IDs, and session references so downstream analytics can trace multi-service flows.

5. **Validate Locally:** Run npm run lint:audit to ensure schema conformance and npm run test:audit to execute instrumentation unit tests.

Probe Management System

Location: /server/src/modules/probes

TL;DR

The Probe Management System orchestrates how evidence-collection probes are registered, deployed, scheduled, and monitored across environments.

It exposes a Probe SDK (implemented in server/src/modules/probes) that standardizes authentication, retries, payload schemas, and version negotiation.

This guide explains lifecycle workflows, API contracts, configuration patterns, and operational practices for integrating probes with enterprise systems such as infrastructure logging platforms, data lakes, and CI/CD pipelines.

1. Purpose and Scope

The Probe Management System enables automated, policy-driven evidence collection for AI governance.

It defines how probes are registered, deployed, scheduled, and monitored, while ensuring data is mapped to the platform's checks and controls for compliance scoring.

This document targets platform engineers, integration teams, and partner developers building or operating probes via the Probe SDK.

2. System Components

2.1 Registry Service

- Persists probe metadata (ID, owner, framework bindings, supported evidence types, current version) in the probes table described in the database design.
- Exposes administrative APIs (/api/probes) for CRUD operations and lifecycle transitions (draft → active → deprecated).

- Issues API credentials and signing keys for each probe instance and enforces RBAC scopes via Casbin policies.
- Stores environment overlays (local, staging, production) so deployments can be generated consistently.

2.2 Deployment Coordinator

- Generates deployment manifests (Docker image refs, runtime arguments, secrets) from registry templates and environment overlays.
- Integrates with the CI/CD pipeline to build probe artifacts, tag them with semantic versions, and push them to the artifact repository.
- Publishes deployment intents to the orchestration plane (Kubernetes Jobs/CronJobs, Lambda functions, or VM agents) through the ProbeDeploymentService utility class.
- Records deployment state transitions (queued, running, succeeded, failed) and links results back to the registry for auditability.

2.3 Scheduler and Execution Plane

- Maintains schedules using a hybrid approach:
- Cron-like recurring tasks stored in the scheduler queue.
- Event-driven triggers derived from webhook subscriptions or CI/CD events.
- On-demand executions initiated through the admin console or API.
- Leverages the ProbeScheduler module to translate schedule definitions into orchestration primitives (Kubernetes CronJob specs, Airflow DAGs, or serverless invocations).
- Provides execution sandboxes with scoped credentials and network policies to isolate probes per tenant and environment.

2.4 Observability and Alerting

- Streams heartbeat, status, and metric events into the observability pipeline (OpenTelemetry collectors feeding Prometheus + Loki).
- Normalizes logs with probe identifiers and control mappings to support search and correlation.
- Emits health signals to the monitoring system, enabling proactive detection of failures or drift.

3. Probe Lifecycle

3.1 Registration Workflow

1. **Proposal:** Engineers register a probe via /api/probes or the admin UI, supplying metadata, evidence schema, and supported frameworks.

2. **Validation:** The ProbeValidationService validates schema definitions, required capabilities, and authentication modes before accepting the probe.

3. **Approval:** Governance admins review the submission, assign ownership, and approve deployment environments.

4. **Credential Issuance:** On approval, the registry issues API keys or mTLS certificates scoped to the probe and environment.

5. **Activation:** The probe transitions to the active state, allowing deployments and schedule binding.

6. **Deprecation:** Deprecated probes retain historical evidence but cannot initiate new runs; replacement probes inherit schedules where applicable.

3.2 Deployment Pipeline

1. **Artifact Build:** Probe source is built via CI, packaged into a container or serverless bundle, and version-tagged (e.g., probe-snowflake@2.1.0).

2. **Manifest Generation:** Deployment Coordinator merges probe defaults with environment-specific overrides (PROBE_ENV, API endpoints, secrets references).

3. **Preflight Checks:** The SDK's ProbeHealthClient executes selfTest() to validate connectivity and schema compliance before rollout.

4. **Rollout:** Scheduler applies manifests to target environments; progress is tracked through deployment events.

5. **Post-Deployment Verification:** Health checks confirm heartbeats and sample payload ingestion; results recorded in the audit log.

6. **Rollback:** If verification fails, the coordinator rolls back to the previous stable version and alerts probe owners.

3.3 Scheduling Strategies

- **Time-Based:** Cron expressions (0 */2 * * *) define periodic evidence collection; stored per environment to reflect differing SLAs.
- **Event-Driven:** SDK exposes registerEventTrigger() to listen to webhook topics (e.g., ci.pipeline.completed) and queue executions.
- **Ad-Hoc:** Investigators can invoke /api/probes/:id/run with parameters for targeted checks or remediation follow-up.

- **Priority Queues:** Scheduler prioritizes runs based on risk tier, regulatory deadlines, or upstream alerts.

4. Probe SDK APIs (server/src/modules/probes)

4.1 Core Classes

Class	*Responsibility*
ProbeClient	Authenticates requests, signs payloads, and exposes submitEvidence() / submitHeartbeat() helpers.
ProbeScheduler	Registers schedules and triggers with the central scheduler service.
ProbeHealthClient	Performs preflight tests, health checks, and dependency diagnostics via selfTest() and reportStatus().
ProbeConfigLoader	Resolves configuration overlays (environment variables, remote config) and merges them into runtime settings.
ProbeVersionManager	Negotiates SDK/runtime versions, enforces minimum compatibility, and exposes getSupportedVersions().

4.2 REST Endpoints

Endpoint	*Method*	*Description*
/api/probes	POST	Register a new probe; accepts metadata, evidence schema, and default schedule definitions.
/api/probes/:probeId	PATCH	Update probe metadata, rotate credentials, or change lifecycle state.
/api/probes/:probeId/deployments	POST	Trigger deployment to a specified environment and version.
/api/probes/:probeId/schedules	PUT	Define or update cron/event schedules tied to framework controls.
/api/probes/:probeId/run	POST	Launch an ad-hoc execution with optional scope filters (system, project, framework).

Endpoint	Method	Description
/api/probes/:probeId/metrics	GET	Retrieve heartbeat metrics, last run status, and failure counts for monitoring dashboards.

4.3 Event Contracts

- **Heartbeat Event (probe.heartbeat.v1):** Emitted by submitHeartbeat(); includes probe ID, version, environment, and latency metrics.
- **Evidence Event (probe.evidence.v1):** Triggered by submitEvidence() after schema validation; contains control mappings, payload hash, and storage references.
- **Failure Event (probe.failure.v1):** Published when retries are exhausted; consumed by alerting workflows for escalation.
- **Deployment Event (probe.deployment.v1):** Broadcast during rollout phases to synchronize UI state and audit logs.

5. Platform Behaviors

5.1 Authentication and Secrets

- Probes authenticate via signed JWTs or mTLS; credentials issued per environment and rotated automatically every 30 days.
- The SDK loads secrets from environment variables or vault integrations (AWS Secrets Manager, HashiCorp Vault).
- All outbound requests sign payloads with SHA-256 HMAC headers; server verifies signatures before accepting evidence.
- Admins can revoke credentials instantly, forcing probes to re-register or fetch updated secrets.

5.2 Retry Semantics

- Network requests implement exponential backoff with jitter (initial delay 2s, max 2m, 5 attempts) and circuit breaker protection.
- Idempotency keys prevent duplicate evidence ingestion during retries.
- Scheduler retries failed runs based on failure class: transient (3 retries), systemic (1 retry, escalate), validation (no retry, mark failed).
- Probes emit failure diagnostics in structured JSON to support forensic analysis.

5.3 Versioning and Compatibility

- Semantic versioning governs both probes (major.minor.patch) and the SDK runtime.
- ProbeVersionManager enforces minimum SDK versions per environment; incompatible probes are blocked during registration.
- Deployments require explicit version targets; canary mode allows n% traffic to the new version before full rollout.
- Deprecated probes remain readable for historical evidence but cannot submit new data once the sunset date passes.

6. Integration Scenarios

6.1 Infrastructure Logs

- Deploy probes adjacent to log aggregation stacks (e.g., CloudWatch, Stackdriver, ELK) to ingest security and access logs.
- Configure collectors to filter events by governance-relevant fields (principal, resource, action) and map them to controls like LOGGING-INTEGRITY.
- Use event-driven schedules triggered by log ingestion pipelines to minimize latency and support near real-time compliance dashboards.

6.2 Data Lakes and Warehouses

- Probes connect to Snowflake, BigQuery, or Delta Lake using read-only service accounts.
- Apply row-level filters and column whitelists defined in the probe configuration to respect data minimization principles.
- Schedule nightly scans to validate retention policies, access controls, and schema drift; map findings to data governance checks.
- Cache metadata locally to reduce load; rely on SDK retries to handle transient warehouse throttling.

6.3 CI/CD Hooks

- Integrate probes into CI/CD pipelines (GitHub Actions, GitLab CI, Jenkins) via webhooks or pipeline tasks.
- Trigger ad-hoc runs when models are promoted, infrastructure templates change, or security scans complete.
- Enforce release gates: pipelines verify probe results (e.g., probe.failureCount == 0) before allowing production deployments.

- Store pipeline context (commit SHA, artifact IDs) with evidence payloads for traceability.

7. Environment Configuration

7.1 Parameterizing Probe Deployments

- Maintain .env.dev, .env.staging, .env.prod overlays for each probe mirroring the platform's environment guide.
- Use the SDK's ProbeConfigLoader to merge shared defaults with environment-specific overrides (API base URLs, tenant IDs, credential paths).
- Annotate deployments with environment labels (env=staging, region=eu-west-1) to aid routing and observability.
- Automate configuration drift detection by comparing deployed manifests against registry templates during health checks.

7.2 Mapping Evidence to Checks and Controls

- During registration, define evidence schemas that include controlId, checkId, and optional riskTier attributes.
- Use the mapping matrix maintained in the registry to bind probes to framework controls (e.g., ISO 42001, NIST RMF).
- SDK provides helper mapEvidenceToControl(controlId, payload) ensuring consistent metadata and storage references.
- Platform ingestion service validates mappings and updates compliance scores in real time when evidence states change.

8. Monitoring Probe Health

8.1 Failure Detection

- Heartbeat intervals (default 5 minutes) monitored via Prometheus; missing two intervals triggers a Warning alert.
- Evidence ingestion errors flagged with structured error codes (EVIDENCE_SCHEMA_MISMATCH, AUTH_EXPIRED).
- Scheduler tracks consecutive run failures; exceeding thresholds escalates severity and notifies owners.

8.2 Alerting Playbooks

- **Tier 1 (Probe Owner):** Receive Slack/Email alerts with run context, logs, and remediation steps.
- **Tier 2 (Platform SRE):** Engaged after 30 minutes of unresolved incidents; access detailed telemetry and can trigger rollbacks.
- **Tier 3 (Governance Lead):** Notified for high-risk control impacts or prolonged outages (>4 hours) affecting regulatory deadlines.
- Integrate alerting with incident management tools (PagerDuty, ServiceNow) using standardized payloads.

8.3 Continuous Improvement Loop

- Post-incident reviews capture root causes, configuration changes, and mitigation tasks.
- Update registry templates with new validation rules or guardrails derived from incidents.
- Feed insights into the roadmap for probe SDK enhancements (improved retries, richer diagnostics).

9. Appendix: Quick Reference Tables

Lifecycle States

State	*Description*	*Allowed Transitions*
draft	Probe registered but awaiting approval; test credentials only.	draft → active, draft → rejected
active	Fully approved and deployable to configured environments.	active → deprecated, active → suspended
suspended	Temporarily disabled (security incident, maintenance).	suspended → active, suspended → deprecated
deprecated	Read-only; retains historical evidence but blocked from new runs.	deprecated → archived
archived	Historical reference only; evidence preserved per retention policy.	archived (terminal)

SDK Utility Methods

Method	*Purpose*
submitEvidence(payload, options)	Sends evidence to /api/probes/:id/evidence with retries and idempotency keys.

Method	*Purpose*
submitHeartbeat(status)	Emits heartbeat events with runtime metrics and version info.
selfTest()	Executes dependency checks (network, credentials, schema validation) before deployment.
registerSchedule(type, config)	Declares cron or event-based schedules managed by the scheduler.
reportStatus(runId, outcome)	Publishes run results for dashboard visibility and alert correlation.

The Probe Management System aligns probe development, deployment, and monitoring with the platform's compliance objectives, ensuring trustworthy evidence collection across diverse enterprise environments.

Check Management System

Location: /server/src/modules/governance/checks

TL;DR

The check management system operationalizes governance requirements inside the Governance Engine.

It coordinates check definitions, execution workflows, evidence capture, and publication workflows across automated and manual paths.

Use this runbook to understand data models, lifecycle states, and operational playbooks for managing compliance checks.

1. Conceptual Overview

The Check Management System is the core capability of the Governance Engine responsible for:

- Converting governance requirements into machine- and human-executable validations.
- Coordinating probe integrations that gather evidence from connected systems.
- Persisting outcomes and evidence metadata so downstream services (dashboards, reporting, remediation) can consume authoritative compliance signals.
- Governing the lifecycle of checks from draft through retirement while preserving audit history and traceability.

Checks always evaluate within the context of a **control** that belongs to a governance framework (e.g., EU AI Act, ISO/IEC 42001). Each check may require one or more **probes** to collect measurements or documentation that substantiate the control’s compliance state.

2. Check Types

Type	*Description*	*Primary Actors*	*Evidence Sources*
Automated	Fully programmatic validations executed by probes against APIs, logs, configuration stores, or model metadata.	Governance Engine scheduler, probe connectors.	API responses, configuration snapshots, log extracts, telemetry metrics.
Manual	Human attestation tasks completed by compliance officers or domain experts when automated evidence is unavailable or requires interpretation.	Governance reviewers, Task Service.	Uploaded documents, interview notes, policy attestations.
Hybrid	Combine automated collection with manual validation to interpret nuanced findings (e.g., bias test results requiring human sign-off).	Governance Engine plus reviewers for exception handling.	Automated probe output supplemented with reviewer commentary or approvals.

Key distinctions:

- **Execution trigger:** Automated checks run on schedules or event hooks, manual/hybrid checks are queued via the Governance Engine after prerequisite evidence is gathered.
- **Evidence structure:** Automated outputs are structured JSON payloads; manual/hybrid entries store attachments and free-form observations linked to evidence records.
- **Approval chain:** Manual/hybrid checks require explicit reviewer sign-off before results are published to controls.

---

3. Execution Workflows

3.1 Automated Checks

1. **Scheduling:** The Governance Engine scheduler polls eligible checks based on cadence metadata (`frequency`, `next_run_at`).

2. **Probe Invocation:** Relevant probe integrations are triggered with context (control ID, environment, target system parameters).

3. **Result Evaluation:** Probe responses are evaluated against rule definitions (thresholds, pattern matches, boolean assertions).

4. **Outcome Recording:** The Engine writes a new row in the results table with computed status, severity, evidence linkage, and raw payload references.

5. **Notifications:** Failed or warning-level results dispatch remediation tasks and alerts through the Notification and Task Services.

3.2 Manual Checks

1. **Task Generation:** Controls flagged for manual validation create queue items assigned to compliance reviewers.

2. **Reviewer Intake:** Reviewers upload evidence or attestations via the Governance UI; metadata is stored in the Evidence Repository and linked to the check.

3. **Assessment:** Reviewers set status, severity, and add narrative evidence references (URLs, document IDs).

4. **Submission:** Results enter a pending_validation state awaiting Governance Engine verification.

5. **Publication:** After validation, the Engine promotes the result to published, making it visible to dashboards and reports.

3.3 Hybrid Checks

1. **Automated Pass:** The Engine executes the automated portion as in section 3.1, marking the result requires_review if human interpretation is necessary.

2. **Reviewer Overlay:** Manual reviewers receive a pre-populated queue item containing probe output, contextual guidance, and recommended next steps.

3. **Finalization:** Reviewers adjust severity/status if needed, append commentary, and resubmit for final validation before publication.

---

4. Data Model and Storage

Checks and outcomes persist in PostgreSQL as part of the Governance Engine schema.

4.1 checks Table (definition layer)

Column	Type	Description
id (PK)	UUID	Unique identifier for the check definition.
control_id (FK)	UUID	Links to the control the check substantiates.
probe_id (FK, nullable)	UUID	References the probe that executes the automated portion; null for purely manual checks.
type	ENUM(automated, manual, hybrid)	Drives workflow selection.
name	Text	Human-readable title shown in the Governance UI.
description	Text	Detailed purpose and validation instructions.
severity_default	ENUM(info, low, medium, high, critical)	Default severity applied when results do not override it.
status	ENUM(draft, active, retired)	Definition lifecycle state.
version	Integer	Incremented whenever validation logic changes.
frequency	Interval	Recommended run cadence for the scheduler.
created_by / updated_by	UUID	User accounts responsible for definition governance.
metadata	JSONB	Free-form configuration for probes (API endpoints, thresholds, mapping rules).

4.2 results Table (execution layer)

Column	Type	Description
id (PK)	UUID	Unique identifier for a specific execution.
check_id (FK)	UUID	Links back to the check definition.
control_id (FK)	UUID	Denormalized reference for reporting joins.
probe_run_id	UUID	Tracks the originating probe execution (if automated).
status	ENUM(pass, fail, warning, pending_validation, requires_review)	Execution outcome.

Column	Type	Description
severity	ENUM(info, low, medium, high, critical)	Impact rating assigned at runtime.
evidence_link_id	UUID	Points to entries in evidence_links that store artefacts and documentation references.
notes	Text	Analyst comments, remediation guidance, or contextual explanations.
executed_at	Timestamp	Actual execution time.
validated_at	Timestamp	Governance Engine validation timestamp.
published_at	Timestamp	When the result became visible to downstream consumers.
created_by	UUID	Reviewer or system user initiating the result.
raw_output	JSONB	Persisted snapshot of probe responses or manual form submissions.

Indexes:

- results_check_id_executed_at_idx for chronological lookups.
- Partial index on status IN ('fail', 'warning') to accelerate remediation queries.
- Composite index on (control_id, published_at) for dashboard aggregations.

---

5. Mapping Checks to Probes and Controls

1. **Controls as Source of Truth:** Every check must map to a single control; the Framework Service maintains the control catalogue and exposes metadata (framework identifier, control category) to the Governance Engine.

2. **Probe Bindings:** Automated or hybrid checks specify a probe_id that resolves to an integration adapter responsible for collecting evidence. Probe definitions include authentication secrets, connection parameters, and supported data domains.

3. **Control Coverage Metrics:** The Governance Engine computes coverage by aggregating active checks per control. Controls can require multiple checks (e.g., technical validation plus policy attestation) to achieve full coverage.

4. **Evidence Linking:** Probe outputs or uploaded documents are stored via the Evidence Repository. results.evidence_link_id anchors evidence objects so auditors can trace findings back to source artefacts.

5. **Status Propagation:** Result status and severity flow upward to control-level scores. Multiple failed checks on a control escalate the control's risk state, triggering governance notifications and remediation tasks.

---

6. Validation and Publishing Lifecycle

Stage	*Definition*	*Trigger*	*Responsible Actor*
draft	Initial definition being authored. Not executable.	Check creation.	Control owner / compliance architect.
ready_for_validation	Logic and metadata completed; awaiting Governance Engine review.	Author submits definition.	Governance Engine governance reviewers.
active	Definition approved and available for scheduling.	Validation approved.	Governance Engine automation.
pending_validation (result)	Execution completed but awaiting secondary review.	Manual/hybrid submissions or automated exceptions.	Governance reviewers.
published (result)	Approved outcome visible to dashboards and reports.	Validator sign-off.	Governance Engine automation.
retired	Definition deprecated; no longer scheduled but retained for audit history.	Replacement created or control removed.	Governance governance board.

Validation steps:

1. **Schema Validation:** Ensure definition metadata includes required fields (control_id, type, severity_default, frequency).

2. **Probe Contract Check:** Automated/hybrid checks must reference an active probe whose schema matches expected payload fields.

3. **Governance Review:** Reviewer confirms mapping to controls, severity rationale, and evidence retention requirements.

4. **Publication:** Upon approval, the Governance Engine updates checks.status to active and queues the first execution.

5. **Ongoing Monitoring:** Failed results trigger revalidation of logic; repeated false positives prompt review of thresholds and severity.

Publishing controls the transition from raw execution to auditor-facing records. Only results with published_at set are included in compliance reports.

---

7. Operational Playbooks

7.1 Adding a New Check

1. **Author Definition:** Draft a YAML or JSON definition (via Governance Engine admin UI) including control mapping, type, severity, frequency, and probe configuration.

2. **Attach Probe (if applicable):** Select an existing probe or request a new integration via the Integration team. Validate connection parameters in a sandbox run.

3. **Submit for Validation:** Move the check to ready_for_validation. Provide supporting rationale and expected evidence samples.

4. **Governance Review:** Governance reviewers test the check in staging, confirm status mappings, and verify evidence storage.

5. **Activate:** Once approved, the check status becomes active and it enters the scheduling queue.

7.2 Version Management

- **Increment version:** Any change to validation logic, severity defaults, or probe bindings requires a version increment. Minor metadata adjustments (description text) do not.
- **Migration Script:** Use the Governance Engine migration pipeline to seed updated definitions; ensure previous versions remain readable for historical results.
- **Backward Compatibility:** Results created under older versions retain their version number; dashboards surface version drift to highlight stale checks.
- **Deprecation:** To replace a check, mark the older version retired after the new version publishes its first successful result, guaranteeing overlap coverage.

7.3 Manual Review Queue Handling

1. **Queue Intake:** Manual and hybrid checks automatically open review tickets in the Governance Engine's Review Queue module with SLA metadata (due date, priority derived from severity).

2. **Assignment:** Queue managers assign reviewers based on expertise and workload; assignments sync with the Task Service for accountability.

3. **Review Execution:** Reviewers follow structured forms capturing attestation statements, evidence references, and severity adjustments. Evidence uploads go through the Evidence Repository.

4. **Validation & Publishing:** Completed reviews transition to pending_validation. Senior reviewers or automated policies (two-person rule) approve for publication, setting validated_at and published_at.

5. **Governance Reporting:** The queue exposes metrics to the Governance Engine dashboard—aging items, SLA breaches, and per-control review cadence—to inform compliance leadership.

Escalation paths are triggered automatically for overdue manual checks, notifying Governance and Risk leads to reassign or adjust severity.

Appendix A. Reference Status and Severity Codes

- **Status Codes:**
- pass – Control requirement satisfied.
- fail – Control requirement breached; remediation required.
- warning – Control partially satisfied; review recommended.
- pending_validation – Awaiting reviewer or automated validation step.
- requires_review – Automated output collected; human action needed before publication.
- **Severity Levels:**
- info – Informational, no immediate action.
- low – Minor gap; monitor.
- medium – Moderate risk; remediation within standard SLA.
- high – Significant risk; expedited remediation.
- critical – Severe risk; immediate executive attention.

These enums align with Governance Engine configuration files and should remain synchronized with backend validation logic.

Control Management System

Location: /server/src/modules/governance/controls

TL;DR

The control management system anchors governance, risk, and compliance posture across the platform.

It maintains a canonical control catalog, maps controls to frameworks and checks, and orchestrates scoring, remediation, and reporting flows.

Use this reference to understand data models, lifecycle procedures, and integrations that keep control health accurate and auditable.

1. Overview

The control management system centralizes definition, evaluation, and reporting for governance, risk, and compliance (GRC) controls. It aligns a shared control taxonomy with multiple regulatory frameworks, maps automated and manual checks to those controls, and drives remediation workflows and dashboard reporting.

2. Control Taxonomy

Controls are organized into a multi-level taxonomy to ensure traceability from enterprise objectives down to individual validation checks.

- **Domain** – Broad thematic area such as Identity & Access, Data Protection, or Infrastructure Security.
- **Category** – Subdivision of a domain that groups controls by functional capability (e.g., Authentication, Encryption).
- **Control** – The authoritative statement specifying required behavior or configuration.
- **Sub-control (optional)** – Additional granularity for complex controls with multiple enforcement mechanisms.

Each control and sub-control must map to at least one framework requirement and one verification check. Domains and categories are maintained in reference tables to preserve consistent naming.

3. Controls Table Metadata

The controls table captures the authoritative record for every control. Core columns include:

Column	*Description*
control_id	Immutable identifier (UUID) used across services.
domain_key	Foreign key to the control domain catalog.
category_key	Foreign key to the control category catalog.
title	Human-readable control name (unique within a domain).
description	Detailed expectation for the control.
rationale	Business or regulatory justification for the control.
implementation_guidance	Prescribed steps or configurations to satisfy the control.
owner_team	Primary team accountable for the control.
status	Lifecycle state (draft, active, deprecated).
risk_tier	Impact rating used in scoring weights (e.g., High/Medium/Low).
version	Integer incremented on every material change.
created_at / updated_at	Audit timestamps maintained by triggers.
created_by / updated_by	User identifiers for change tracking.

3.1 Relationships to Frameworks

Framework mappings live in a control_framework_links join table linking control_id to framework_id and requirement_id. Each mapping includes:

- coverage_level – Degree of fulfillment (e.g., Full, Partial, Compensating).
- evidence_reference – Link to policy or artifact proving compliance.
- effective_date / expiry_date – Validity window for the mapping.

3.2 Relationships to Checks

Verification checks reside in the checks table and connect to controls via the control_check_links join table. Each link records:

- check_id – Identifier for the automated or manual assessment.
- assertion_type – Nature of evaluation (configuration, process, interview, etc.).
- frequency – Expected cadence for running the check.
- enforcement_level – Whether failure blocks deployments, triggers incidents, or only surfaces in reports.

4. Scoring and Status Determination

Control posture scores combine check outcomes, weighting, and risk tiers:

1. Each linked check produces a normalized score between 0 and 1 based on raw findings (e.g., pass = 1, warning = 0.5, fail = 0).

2. Check scores are multiplied by their weight (defined per control-check link) and aggregated.

3. The aggregate score is scaled by the control's risk_tier multiplier (High = 1.5, Medium = 1.0, Low = 0.75 by default).

4. The final control score is capped at 1.0. Thresholds classify status: >=0.85 = Passing, 0.60-0.84 = Needs Attention, ≤0.60 = Failing.

4.1 Handling Failures

When a check fails:

- The control's status transitions to Failing if the recalculated score drops below threshold.
- An incident ticket and remediation task are created automatically when the enforcement level is "Blocking" or "Critical".
- Evidence of the failure, including check output and timestamps, is persisted for audit.
- Notification rules alert the owner_team and stakeholders subscribed to the impacted framework.

4.2 Linkage to Remediation Tasks and Dashboards

Control scores feed dashboards showing posture by domain, framework, and owner team. Remediation tasks include references to control_id, failing check_id, and due dates derived from risk tier. Dashboards surface:

- Current score and historical trend per control.
- Open remediation tasks, their status, and SLA adherence.

- Framework coverage heatmaps showing how failures affect compliance objectives.

5. Control Lifecycle Procedures

5.1 Creating a New Control

1. Author draft content in a change request including title, description, rationale, guidance, and initial taxonomy placement.

2. Validate deduplication against existing controls and align with relevant frameworks.

3. Submit the draft to the governance board for review; reviewers approve domain/category and risk tier.

4. Upon approval, insert the new record into the controls table with status = 'active' and create mandatory framework and check mappings.

5. Schedule initial checks and confirm dashboard visibility.

5.2 Updating an Existing Control

1. Initiate a versioned change request documenting proposed modifications and impacted frameworks.

2. Update the control record, incrementing the version and capturing change notes in the audit log.

3. Review and adjust linked checks, weights, and remediation workflows to reflect the new requirements.

4. Notify dependent teams and update documentation or playbooks.

5.3 Mapping Controls Across Frameworks

1. Identify equivalent requirements in target frameworks using the taxonomy reference matrix.

2. Create or update entries in control_framework_links with coverage level, evidence references, and validity dates.

3. Ensure compensating controls are documented when full coverage is not achievable.

4. Recalculate framework posture dashboards to reflect the new mappings.

5.4 Auditing Changes

- All inserts and updates to controls, control_framework_links, and control_check_links trigger audit log entries capturing user, timestamp, before/after snapshots, and approval ticket references.
- Quarterly audits review a random sample of controls to verify metadata accuracy, evidence freshness, and mapping completeness.
- Any discrepancies are logged as remediation tasks with defined owners and due dates.

- Audit findings feed into leadership dashboards and compliance reports.

6. Appendices

- **Reference Tables**: Domain catalog, category catalog, framework registry.
- **Key Integrations**: Incident management (for blocking failures), task tracking (remediation assignments), business intelligence (dashboarding).
- **Service Interfaces**: REST endpoints for control CRUD operations, webhook for check result ingestion, and data warehouse exports for analytics.

Framework Mapping System

Location: /server/src/modules/frameworks

TL;DR

The framework mapping system governs lifecycle management for regulatory frameworks and their control mappings.

It centralizes metadata, aligns controls across standards, and exposes APIs for onboarding, versioning, and exporting frameworks.

Use this guide to understand data contracts, CRUD flows, and operational guardrails that keep mappings authoritative.

1. Module Overview

The **Framework Mapping System** in server/src/modules/frameworks standardizes how governance frameworks, controls, and cross-framework mappings are stored and exposed through the API. It follows the layered architecture used across the backend:

```
server/src/modules/frameworks
■■■ controllers/ → HTTP handlers (Express routers)
■■■ dto/ → Request/response contracts validated with Zod
■■■ services/ → Business logic orchestrating repositories and events
■■■ repositories/ → Prisma queries for frameworks, controls, mappings, versions
■■■ mappers/ → Mapping utilities for entity ↔ DTO transformations
■■■ policies/ → Casbin-powered authorization guards
■■■ tasks/ → BullMQ queues for imports, exports, version diffing
```

■■■ subscribers/ → Domain events for scoring + reporting recalculations

■■■ index.ts → Module bootstrap, router registration, dependency wiring

```

The module exposes REST + event-driven interfaces that allow:

- **Framework CRUD:** create, read, update, soft-delete, and restore frameworks.
- **Control catalogs:** maintain framework-specific controls with scoped metadata.
- **Cross-mapping:** map controls across internal and external frameworks for "comply once, satisfy many" workflows.
- **Version governance:** snapshot framework revisions, track historical mappings, and trigger score recalculations.
- **Synchronization hooks:** emit events consumed by reporting, evidence, and notification modules whenever mappings change.

## 2. Data Model & Metadata Contracts

| *Entity* | *Table* | *Key Fields* | *Notes* |
|---|---|---|---|
| Framework | frameworks | id, slug, title, version, domain, jurisdiction, publisher, valid_from, valid_to, status, metadata (JSONB) | Metadata stores arbitrary key/value pairs such as regulatory references, risk tier, and localization tags. slug is globally unique for routing and import/export alignment. |
| Control | controls | id, framework_id, code, title, description, category, risk_level, evidence_requirements, metadata | Controls belong to exactly one framework version. code is unique per framework. |
| Mapping | mappings | id, source_framework_id, source_control_id, target_framework_id, target_control_id, mapping_strength, justification, tags, metadata, status | Supports many-to-many relationships between controls. mapping_strength uses enums (exact, partial, informative). |
```

Entity	Table	Key Fields	Notes
FrameworkVersion	framework_versions	id, framework_id, major, minor, patch, change_log, published_at, approved_by, diff_hash, metadata	Tracks snapshots of framework + mapping state. Stored as semver to allow compatibility rules.
ImportBatch	framework_imports	id, source, format, status, submitted_by, payload_uri, processed_at, error_report_uri	Used for asynchronous ingestion of CSV/JSON imports.
ExportJob	framework_exports	id, format, filters, requested_by, status, artifact_uri, expires_at	Supports scheduled exports and audit evidence.

Metadata Schema Highlights

- **Global identifiers:** metadata.external_ids (list) keeps regulatory IDs and links to authoritative sources.
- **Localization:** metadata.localization.{locale}.title enables translated labels.
- **Scoring hints:** metadata.weighting (0–1) influences aggregated compliance scoring.
- **Lifecycle flags:** metadata.lifecycle.stage (draft, active, sunset) and metadata.lifecycle.review_due for governance SLAs.
- **Data lineage:** metadata.source_hash ensures reproducibility of imported frameworks.

3. CRUD Flows

3.1 Framework Lifecycle

Step	Action	Component	Notes
1	POST /frameworks with CreateFrameworkDto	FrameworkController.create → FrameworkService.create	Validates payload (Zod), enforces Casbin policy (frameworks:create). Generates slug, persists framework + initial version (v1.0.0) inside a transaction. Emits framework.created event.
2	GET /frameworks	FrameworkController.list → FrameworkService.list	Supports filtering by jurisdiction, lifecycle stage, search, pagination. Returns DTO containing active version and summary counts.
3	GET /frameworks/:id	FrameworkController.get	Fetches framework with latest version, control counts, mapping coverage stats.
4	PATCH /frameworks/:id	FrameworkController.update → FrameworkService.update	Partial updates; tracks changed fields for audit log. When version-affecting fields change (title, metadata.lifecycle), service creates a draft revision.
5	DELETE /frameworks/:id	FrameworkController.archive	Soft deletes; sets status = archived and triggers framework.archived. Controls and mappings remain for historical reports but are hidden in active queries.
6	POST /frameworks/:id/restore	FrameworkController.restore	Restores archived frameworks, revalidates slug collisions, and replays cached mappings.

3.2 Control Registration

1. POST /frameworks/:id/controls

- Validated by CreateControlDto (code uniqueness enforced).
- Service associates control with the draft or active version depending on payload flag attachToDraft.
- Emits control.created and recalculates framework coverage metrics.

2. PATCH /frameworks/:id/controls/:controlId

- Supports editing metadata, evidence requirements, and risk levels.
- When breakingChange = true, service triggers a new draft version and adds diff entry.

3. DELETE /frameworks/:id/controls/:controlId

- Marks control as deprecated; mapping service downgrades mapping strength to informative to avoid orphaned dependencies.

4. Bulk upload via POST /frameworks/:id/controls/import

- Accepts CSV/JSON zipped artifacts.
- Stored in framework_imports and processed asynchronously via FrameworkImportTask (BullMQ). Validation errors compiled into downloadable report.

3.3 Mapping Management

- **Create mapping:** POST /frameworks/:id/mappings expects CreateMappingDto containing source control, target framework/control, strength, justification, tags.
- Service ensures both controls exist and are version-compatible.
- Creates symmetrical edge if biDirectional = true.
- Emits mapping.created event consumed by scoring + reporting microservices.

- **Update mapping:** PATCH /frameworks/:id/mappings/:mappingId
- Allows adjusting mapping_strength, tags, justification, or re-pointing to a new target control.
- Maintains change history in mapping_history table for audit.

- **Delete mapping:** soft delete via DELETE to preserve version history. Related exports mark mapping as status = retired.

- **Bulk operations:** POST /frameworks/:id/mappings/import

- Supports multi-framework CSV/JSON format with columns for source/target codes, jurisdiction, and weighting.
- Import pipeline cross-validates codes and warns on missing dependencies.

3.4 Version Governance

1. **Draft creation:**

- POST /frameworks/:id/versions/draft clones current version metadata, copies controls + mappings into framework_version_items table.
- Draft is isolated; updates to controls/mappings flagged with draftVersionId.

2. **Change review:**

- FrameworkVersionService.reviewDraft aggregates diff summary (added/removed controls, mapping strength changes).
- Compliance officers annotate diff and attach evidence or regulatory references.

3. **Approval:**

- POST /frameworks/:id/versions/:versionId/approve
- Locks the draft, increments semver based on diff classification (major/minor/patch).
- Emits framework.version.published event to trigger recalculations and exports.

4. **Rollback:**

- POST /frameworks/:id/versions/:versionId/rollback reactivates previous version, reassigns active mappings, and notifies downstream systems.

5. **Diff export:**

- GET /frameworks/:id/versions/:versionId/diff returns JSON + CSV summarizing changes for auditors.

4. Multi-Framework Compliance Support

4.1 Metadata Alignment Layer

- **Framework taxonomy:** Jurisdiction, sector, risk tier, and lifecycle fields allow filtering frameworks for composite compliance packages.

- **Control tags:** metadata.tags (e.g., {"category": "Transparency", "pillar": "NIST-GOV"}) align controls across frameworks.
- **Equivalence matrix:** Stored in mappings table linking controls across frameworks; enriched with mapping_strength and justification for audit traceability.
- **Coverage analytics:** FrameworkService.get returns coverageMatrix (percentage of source controls mapped per target framework) enabling compliance dashboards to highlight gaps.

4.2 Import & Export Pipelines

- **Import formats:**
- **CSV:** Column headers framework_code, control_code, target_framework_slug, target_control_code, strength, justification, metadata.*.
- **JSON:** Nested structure aligning with DTOs, supporting metadata injection.
- **Regulatory payloads:** Integration with official sources (e.g., EU AI Act) via connectors stored in metadata.source_uri.

- **Validation stages:**

1. Schema validation using Zod.
2. Referential integrity check against existing frameworks/controls.
3. Policy enforcement (only compliance officers can import global mappings).
4. Dry-run preview endpoint POST /frameworks/:id/mappings/import/preview returning diff summary.

- **Export options:**
- GET /frameworks/:id/export?format=csv|json|xlsx generates static snapshots for auditors.
- Multi-framework export includes mapping matrix and localization fields, enabling offline compliance reviews.
- Exports are versioned; artifact metadata contains framework_version_id and mapping_hash for traceability.

4.3 Update Governance

- **Change approval:** All mapping updates require review when affecting more than configurable threshold (e.g., >20 mappings). Workflows integrate with notification service for approvals.
- **Audit trails:** Each mutation writes to framework_audit_log capturing actor, timestamp, payload diff, and resulting version.

- **Dependency notifications:** Reporting service listens to mapping.updated events to refresh cached scoring. If unresolved dependencies exist, service raises tasks for remediation.
- **Backward compatibility rules:**
- Major version increases trigger background job to clone reports and maintain references to previous mappings.
- Soft-deleted mappings remain accessible via status = retired for historical queries.
- ReportService respects effective_from and effective_to fields, so historical reports maintain accurate framework context.

5. End-to-End Examples

5.1 Onboarding a New Framework

1. **Create framework:** Compliance officer invokes POST /frameworks with metadata (jurisdiction, publisher, lifecycle stage).

2. **Seed controls:** Upload CSV through /controls/import. System validates codes, attaches to draft version, and logs warnings.

3. **Draft mappings:** Use POST /mappings/import/preview to align new controls with existing frameworks (e.g., NIST AI RMF).

4. **Review draft version:** Stakeholders review diff summary, add justifications, and approve via /versions/:id/approve (minor version if no breaking changes).

5. **Trigger exports:** Auto-generated export job creates CSV/JSON artifacts for partner distribution.

6. **Notify downstream systems:** framework.version.published event prompts reporting module to compute baseline scores for the new framework.

5.2 Updating Mappings After a Regulatory Change

1. **Monitor regulatory feed:** External connector updates metadata.source_hash, flagging impacted controls.

2. **Bulk mapping edit:** Compliance officer submits CSV with new target control references.

3. **Dry-run validation:** /mappings/import/preview returns warnings for deprecated controls; officer resolves issues and resubmits.

4. **Governance approval:** Because >20 mappings changed, workflow routes draft to governance board. Approvers review diff, add justification referencing new regulation clause.

5. **Publish minor version:** /versions/:id/approve increments minor version, activates new mappings.

6. **Automated recalculation:** Scoring service consumes mapping.updated events, recalculates framework overlap, and posts summary to Slack via notification module.

5.3 Preserving Backward Compatibility in Reports

1. **Report snapshot creation:** When a report is generated, it stores framework_version_id and mapping_hash used at runtime.

2. **New major release:** Framework team introduces major changes (control renumbering). Approval publishes v2.0.0.

3. **Compatibility bridge:** Version service spawns BackwardCompatibilityTask copying retired mappings to compatibility_overrides table with effective_to = NULL.

4. **Historical report rendering:** Report API checks if requested version differs from latest; if so, it loads compatibility overrides and displays retired control codes with status = legacy badge.

5. **Sunset notification:** When effective_to reached, background job notifies report owners to refresh assessments under the new version.

6. Operational Considerations

- **Access Control:** Casbin policies ensure only users with role = compliance_officer or governance_admin can modify frameworks or publish versions. Read access is granted to auditors and product owners.
- **Performance:** Repository layer uses batched Prisma transactions and caching for common lookups (frameworkBySlug, mappingMatrixByFramework). Redis-backed cache invalidated via events.
- **Observability:** Every controller logs structured events (pino) with correlation IDs. BullMQ tasks expose Prometheus metrics for import/export throughput and failure rates.
- **Error Handling:** Controllers wrap service errors into standardized ApiError objects with machine-readable codes (FRAMEWORK_NOT_FOUND, MAPPING_VALIDATION_FAILED).
- **Testing:** Module includes Jest suites for service logic, contract tests for controllers, and snapshot tests for diff exports. Integration tests run via Postman collection frameworks.postman.json during CI.

Evidence Management System

Location: /server/src/modules/evidence

TL;DR

The evidence management system governs how compliance artifacts are collected, secured, and distributed.

Implemented in server/src/modules/evidence, it orchestrates presigned upload/download flows, metadata persistence, and immutable audit trails.

This reference documents ingestion patterns, schema design, operational safeguards, and integrations with controls, checks, and tasks.

1. Module Overview

The evidence management capability lives under server/src/modules/evidence and provides the backend surface for collecting, securing, and distributing compliance artifacts across the platform. It integrates with the shared MinIO client in server/src/integrations, exposes dedicated upload, download, and metadata endpoints, and enforces encryption, presigned URL workflows, and linkage back to controls and tasks.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L60-L133■

2. Upload and Download Flows

2.1 Manual and Assisted Uploads

1. A privileged user (e.g., Compliance Officer) requests an upload session from /evidence/upload.

2. The module generates a short-lived presigned URL that allows the browser or probe to stream the file directly into the externally hosted MinIO bucket without routing through the API server, while simultaneously preparing the metadata payload that will be persisted in PostgreSQL once the transfer succeeds.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L122-L133■■F:docs/02-technical-specifications/01-system-architecture.md†L166-L193■

3. On completion, the module captures file attributes (size, checksum, MIME type), the initiating user or system identity, and the governance objects supplied in the upload request (control ID, check ID, task ID). This metadata is stored in the evidence and evidence_links tables so that downstream scoring, dashboards, and remediation workflows can resolve relationships immediately.■F:docs/02-technical-specifications/04-database-design.md†L86-L99■

2.2 Automated Probe Ingestion

1. Probes or scheduled collectors fetch raw governance data and call the same upload endpoint with system credentials issued through Casbin policies.

2. Evidence objects emitted by probes inherit the same metadata schema and version tracking, ensuring parity between automated and manual submissions. Probe activity is logged alongside user activity so that audit trails retain the origin of every artifact.■F:docs/02-technical-specifications/04-database-design.md†L80-L99■■F:docs/01-about/04-security-and-data-protection.md†L263-L287■

2.3 Secure Downloads

1. When a reviewer initiates /evidence/:id/download, the module validates RBAC policies, then generates a presigned URL scoped to the object key, HTTP verb, and expiration time.

2. Downloads reference the persisted metadata for size and checksum validation, and the access event is appended to immutable audit logs for traceability.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L122-L133■■F:docs/01-about/04-security-and-data-protection.md†L263-L311■

3. Metadata and Schema Management

Evidence metadata is normalized in PostgreSQL via the evidence table, which stores identifiers, storage object references, version markers, and lifecycle timestamps. Relational tables such as evidence_links associate each record with controls, checks, and remediation tasks, while indexing strategies enable efficient search and reporting across large evidence sets.■F:docs/02-technical-specifications/04-database-design.md†L86-L113■ Integrity constraints and soft-delete strategies prevent orphaned records and guarantee referential accuracy even as evidence ages or is superseded.■F:docs/02-technical-specifications/04-database-design.md†L122-L130■

4. Audit Trails and Monitoring

All interactions with evidence—uploads, approvals, downloads, and metadata edits—are captured in the platform's immutable logging pipeline. Entries record timestamps, actor identities, affected entities, and source network information, and they are hashed and stored in append-only repositories so that tampering is detectable. Retention and archival controls maintain audit readiness while balancing storage efficiency.■F:docs/01-about/04-security-and-data-protection.md†L263-L312■ The evidence module emits structured events that feed these logs, ensuring every compliance artifact maintains a verifiable chain of custody.■F:docs/01-about/04-security-and-data-protection.md†L87-L107■

5. Versioning, Tagging, and Retention Policies

The repository retains a complete, versioned history for each artifact, preserving immutable records of prior submissions and their validation outcomes. Evidence can be tagged by product, control, or framework to support filtered investigations and reporting views across the compliance dashboards.■F:docs/01-about/03-concept-summary.md†L150-L158■ Retention defaults to 36 months, with automated archival to cold storage and GDPR-compliant deletion workflows for records that exceed contractual lifespans or are subject to erasure requests.■F:docs/02-technical-specifications/04-database-design.md†L158-L161■■F:docs/01-about/04-security-and-data-protection.md†L140-L150■ Backup policies extend durability, with nightly full snapshots, hourly differentials, and AES-256 encryption applied before storage.■F:docs/02-technical-specifications/04-database-design.md†L147-L171■

6. Linkage to Controls, Checks, and Tasks

Evidence objects are first-class participants in the governance data model: controls aggregate related checks and attach remediation tasks, each of which references supporting evidence. The evidence_links join table enables one-to-many relationships between controls and artifacts, as well as many-to-one mappings from tasks back to the evidence that validates remediation completion. Task records maintain direct pointers to associated evidence so that closing a remediation item automatically updates compliance status and audit reports.■F:docs/02-technical-specifications/04-database-design.md†L86-L99■■F:docs/01-about/03-concept-summary.md†L124-L158■■F:docs/01-about/03-concept-summary.md†L326-L358■

7. Operational Guidance

7.1 Storage Configuration

Set the MinIO endpoint, access key, and secret key through environment variables managed per environment (.env.dev, .env.staging, .env.prod). Store the secret material in a dedicated vault and inject it during CI/CD deploys. Post-deployment checks must confirm data persistence against PostgreSQL and MinIO, and smoke tests should exercise evidence upload flows before sign-off.■F:docs/02-technical-specifications/08-deployment-and-environment-guide.md†L100-L205■ Bucket lifecycle management and MinIO object versioning are part of the infrastructure-as-code layer, providing scale-out capacity and rollback options for artifacts.■F:docs/02-technical-specifications/05-devops-infrastructure.md†L200-L214■

7.2 Encryption and Key Management

MinIO storage and the backing PostgreSQL metadata store enforce AES-256 encryption at rest and TLS for transit, with keys rotated through managed KMS services. The evidence module inherits these guarantees and validates that presigned URLs are scoped to HTTPS endpoints, ensuring uploaded and downloaded content remains encrypted end-to-end.■F:docs/02-technical-specifications/01-system-archit

ecture.md†L166-L193■■F:docs/01-about/04-security-and-data-protection.md†L87-L134■■F:docs/02-technical-specifications/04-database-design.md†L169-L172■

7.3 Integrity Verification

Cryptographic integrity checks accompany evidence ingestion and logging, enabling the platform to detect tampering across both storage and audit layers. Immutable log storage with hashing, combined with checksum validation from MinIO and the metadata store, ensures that every evidence download can be cross-verified against the original upload record.■F:docs/01-about/04-security-and-data-protection.md†L87-L107■■F:docs/01-about/04-security-and-data-protection.md†L272-L311■ Regular deployment rituals include verifying evidence upload workflows and confirming that monitoring surfaces catch anomalies, maintaining continuous assurance over evidence integrity.■F:docs/02-technical-specifications/08-deployment-and-environment-guide.md†L182-L205■

Governance Engine

Location: /server/src/modules/governance

TL;DR

The governance engine is the platform core that turns evidence into actionable compliance intelligence.

Located at server/src/modules/governance, it executes checks, aggregates control and framework scores, and drives remediation workflows.

This guide explains lifecycle orchestration, execution internals, extensibility hooks, and system dependencies.

1. Module Overview

- **Location:** server/src/modules/governance sits inside the feature-oriented backend directory layout, alongside peers for integrations, middleware, and routes.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L55-L63■
- **Purpose:** The module is the core compliance processor that ingests evidence, evaluates checks, aggregates control health, scores frameworks, and produces risk intelligence.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L90-L102■■F:docs/01-about/03-concept-summary.md†L214-L302■
- **Primary responsibilities:**
- Execute governance checks defined against regulatory or internal policies.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L92-L103■■F:docs/01-about/03-concept-summary.md†L243-L252■
- Aggregate check outcomes into controls and framework-aligned scorecards for reporting and decision making.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L92-L103■■F:docs/01-about/03-concept-summary.md†L255-L279■■F:docs/02-technical-specifications/04-database-design.md†L91-L94■
- Trigger remediation tasks, notifications, and audit logging to ensure issues are triaged, resolved, and permanently recorded.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L9

5-L159■■F:docs/01-about/03-concept-summary.md†L282-L359■■F:docs/02-technical-specifications/04-database-design.md†L96-L99■

2. End-to-End Lifecycle

The governance engine orchestrates a closed-loop workflow that starts with probe data and ends with continuous monitoring. The flow below combines asynchronous ingestion, synchronous validation, and feedback-driven remediation:

```mermaid
flowchart TD
A[Probe ingestion] --> B[Evidence validation]
B --> C[Check execution]
C --> D[Control aggregation]
D --> E[Framework scoring]
E --> F{Compliant?}
F -- Yes --> G[Dashboards & scorecards]
F -- No --> H[Remediation tasks]
H --> I[Task closure & evidence upload]
I --> J[Re-run checks]
J --> D
G --> K[Continuous monitoring]
K --> A
```

2.1 Lifecycle Narrative

1. **Probe ingestion** – Probes built with the SDK collect configuration, audit, and testing data from customer environments and push structured payloads into the governance engine via authenticated endpoints.■F:docs/02-technical-specifications/07-integration-architecture.md†L68-L90■■F:docs/01-about/03-concept-summary.md†L223-L240■

2. **Evidence validation & normalization** – Incoming payloads are authenticated, schema-validated, and mapped to the correct framework/control context before storage in the evidence repository and relational tables (e.g., probes, checks, results).■F:docs/02-technical-specifications/04-database-design.md†L80-L94■

3. **Check execution** – Each control's checks run automatically (or require human approval for manual/hybrid cases), producing compliant / non-compliant / partial outcomes recorded alongside evidence references.■F:docs/01-about/03-concept-summary.md†L243-L252■■F:docs/02-technical-specifications/04-database-design.md†L80-L89■

4. **Control aggregation** – The engine aggregates check outputs to compute control compliance percentages, risk ratings, and maturity indicators, surfacing them on dashboards and reports.■F:docs/01-about/03-concept-summary.md†L255-L279■■F:docs/01-about/03-concept-summary.md†L314-L384■

5. **Framework scoring** – Controls are mapped to multiple frameworks, enabling a single result set to drive EU AI Act, ISO 42001, or NIST AI RMF scorecards. Aggregate scores, heatmaps, and reports are written to the scores and metrics tables for analytics consumption.■F:docs/01-about/03-concept-summary.md†L262-L279■■F:docs/02-technical-specifications/04-database-design.md†L91-L94■

6. **Remediation & tasking** – Failed checks and controls generate remediation tasks with due dates, evidence links, and escalation rules. Task completion triggers control revalidation, closing the feedback loop and updating dashboards and integrations (Jira, ServiceNow, Slack).■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L95-L159■■F:docs/01-about/03-concept-summary.md†L282-L359■■F:docs/02-technical-specifications/07-integration-architecture.md†L95-L141■

7. **Monitoring & auditability** – Continuous probe execution re-runs checks, refreshes scores, and appends audit logs, ensuring posture reflects the current production state and remains export-ready for audits or partner APIs.■F:docs/01-about/03-concept-summary.md†L290-L302■■F:docs/02-technical-specifications/04-database-design.md†L96-L99■■F:docs/02-technical-specifications/07-integration-architecture.md†L145-L165■

3. Execution Model inside server/src/modules/governance

3.1 Check Runners

- **Input sources:** Probe events, manual evidence uploads, and webhook triggers resolved to check records with typed payloads and timestamps.■F:docs/02-technical-specifications/04-database-design.md†L80-L89■
- **Validation pipeline:**

1. Authenticate request and attach tenant + framework metadata.

2. Apply schema validation and normalization (e.g., severity enums, timestamp coercion).

3. Invoke check-specific logic (scripted rule, threshold comparison, or manual review queue).

4. Persist results row and link to evidence entries for traceability.■F:docs/02-technical-specifications/04-database-design.md†L80-L99■

- **Outcome logging:** Every execution writes to the audit log subsystem with actor, source, and outcome to support investigations and compliance attestations.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L95-L103■■F:docs/02-technical-specifications/04-database-design.md†L96-L99■

3.2 Control Aggregation

- **Rollup mechanics:** Controls subscribe to a set of checks; the engine computes compliance percentages by weighting pass/fail states, severity, and freshness of evidence. Results are persisted to the scores table and surfaced in dashboards and exports.■F:docs/01-about/03-concept-summary.md†L255-L279■■F:docs/02-technical-specifications/04-database-design.md†L91-L94■
- **Risk categorization:** Controls produce risk ratings (e.g., critical/major/minor) feeding observations, gap analyses, and remediation priorities.■F:docs/01-about/03-concept-summary.md†L326-L347■
- **Framework mapping:** Mappings maintained by the Framework Service let one control satisfy multiple obligations. The governance engine reuses these relationships when generating framework-specific views or partner API payloads.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L106-L118■■F:docs/01-about/03-concept-summary.md†L262-L279■

3.3 Remediation Orchestration

- **Task generation:** Failed controls trigger records in the Task Service with status, ownership, due dates, and escalation metadata; evidence links tie remediation artifacts back to the originating control.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L95-L165■■F:docs/01-about/03-concept-summary.md†L282-L359■
- **Workflow automation:** Integrations send tickets or alerts to ServiceNow, Jira, or Slack, keeping external systems synchronized during remediation and approvals.■F:docs/02-technical-specifications/07-integration-architecture.md†L95-L141■
- **Closure & revalidation:** Task closure triggers re-execution of linked checks; success updates control scores and clears open observations for audit records.■F:docs/01-about/03-concept-summary.md†L282-L359■■F:docs/02-technical-specifications/04-database-design.md†L96-L99■

4. Logging, Reporting, and Audit Trails

- **Audit logs:** Every governance operation (probe ingestion, check outcome, control update, task state change) appends immutable entries with actor, timestamp, and payload summary for downstream analytics and audits.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L95-L103■■F:docs/02-technical-specifications/04-database-design.md†L96-L99■
- **Score & metric storage:** Control and framework scores, trend metrics, and dashboard KPIs are persisted for historical analysis and visualizations consumed by the reporting layer.■F:docs/01-about/03-concept-summary.md†L269-L384■■F:docs/02-technical-specifications/04-database-design.md†L91-L94■
- **Evidence repository links:** Each result references evidence artifacts housed in MinIO and relational metadata tables, guaranteeing traceability from every score back to raw proof.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L121-L134■■F:docs/02-technical-specifications/04-database-design.md†L86-L89■

- **Partner API and exports:** The module exposes scorecards, audit logs, and compliance data through role-scoped APIs, enabling regulators, partners, and enterprise data lakes to consume governance telemetry securely.■F:docs/02-technical-specifications/07-integration-architecture.md†L145-L165■

5. Extensibility Hooks

5.1 Adding New Checks or Probes

1. **Define probe data contract** using the Probe SDK (fields, authentication, retries) to send evidence payloads to the governance engine.■F:docs/02-technical-specifications/07-integration-architecture.md†L68-L90■

2. **Register check metadata** in the checks table, linking it to the relevant control, probe, and framework mappings.■F:docs/02-technical-specifications/04-database-design.md†L80-L94■

3. **Implement check logic** within the governance module (new validator, rule script, or manual review workflow) and map it to routing/middleware entries under server/src/routes.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L55-L103■

4. **Update control configuration** to include weighting, severity, and notification preferences so aggregated scoring reflects the new signal.■F:docs/01-about/03-concept-summary.md†L255-L347■

5. **Extend integration hooks** (optional) to push failures into task or notification systems for remediation tracking.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L95-L165■■F:docs/02-technical-specifications/07-integration-architecture.md†L95-L141■

5.2 Adjusting Control Weightings and Scoring Logic

- **Control metadata:** Modify control definitions to adjust severity, weight, or maturity thresholds; changes immediately influence rollups in the scores table and downstream dashboards.■F:docs/01-about/03-concept-summary.md†L255-L384■■F:docs/02-technical-specifications/04-database-design.md†L91-L94■
- **Framework mappings:** Update mapping records so adjusted controls cascade to every linked framework requirement without duplicative effort.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L106-L118■
- **Reporting calibration:** Reconfigure dashboard widgets or partner API filters to surface new weighting logic or thresholds for auditors and business stakeholders.■F:docs/01-about/03-concept-summary.md†L269-L384■■F:docs/02-technical-specifications/07-integration-architecture.md†L145-L165■

6. Dependencies and System Interactions

- **Probes:** Supply raw evidence via the Probe SDK and integrations, forming the input stream for checks and controls.■F:docs/02-technical-specifications/07-integration-architecture.md†L68-L118■ ■F:docs/01-about/03-concept-summary.md†L223-L240■
- **Checks:** Encapsulate validation logic stored alongside probes and controls, with outcomes feeding aggregation and logging workflows.■F:docs/02-technical-specifications/04-database-design.md†L80-L89■■F:docs/01-about/03-concept-summary.md†L243-L252■
- **Controls & Frameworks:** Provide the aggregation and mapping layers that translate raw check data into cross-framework compliance insights.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L90-L118■■F:docs/01-about/03-concept-summary.md†L255-L279■
- **Tasks & Notifications:** Downstream services consume governance engine events to create, assign, and escalate remediation tasks while alerting stakeholders through channels like Jira, ServiceNow, and Slack.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L95-L159■■F:docs/02-technical-specifications/07-integration-architecture.md†L95-L141■
- **Evidence Repository & Audit Logs:** Ensure every compliance decision is backed by immutable records accessible to auditors and partner systems.■F:docs/02-technical-specifications/02-backend-architecture-and-apis.md†L121-L134■■F:docs/02-technical-specifications/04-database-design.md†L86-L99■

Together, these components enable the governance engine to deliver continuous, measurable, and auditable AI compliance across the platform.

Task Management System

Location: /server/src/modules/tasks

TL;DR

The task management system converts governance signals into actionable remediation work.

It orchestrates task lifecycle states, assignments, escalations, and evidence capture across internal and external tooling.

Completed tasks automatically trigger control re-validation to maintain continuous compliance.

1. Purpose and Context

The task management system is the operational core of remediation activities. It links failed controls, audit findings, and ad-hoc governance requests to accountable owners, ensuring transparent tracking of remediation progress. Tasks act as the bridge between detection events (e.g., failed automated checks) and verification of corrective actions.

2. Task Creation Triggers

Tasks can be generated through multiple automated and manual channels:

Trigger Source	*Description*	*Default Severity*	*Auto Assignment*
Failed control check	Automated scanners flag control non-compliance.	Based on control risk rating.	Assigned to control owner; fallback to compliance queue.
Risk alert escalation	Alerts raised by monitoring pipelines or anomaly detection that require follow-up.	Critical if alert severity ≥ High.	Assigned to alert responder group.

Trigger Source	Description	Default Severity	Auto Assignment
Audit finding	Internal/external auditors log remediation items during assessments.	Medium unless overridden by auditor.	Assigned to compliance officer responsible for audit scope.
Manual task creation	Users create tasks from dashboard for process changes or documentation updates.	User-selected.	Assigned manually or queued for triage.
External integration	Jira/ServiceNow issues synced into platform when linked to a control.	Mirrors source system priority.	Owner derived from external assignee mapping.

Additional triggers (e.g., policy updates) can be configured via automation rules that map events to task templates.

3. Status Lifecycle

Task status values are standardized to reflect remediation maturity:

1. **Open** – Task logged but not yet acknowledged.
2. **In Progress** – Work has started; owner actively addressing remediation.
3. **Blocked** – Dependencies prevent progress; waiting on external action.
4. **Ready for Review** – Owner completed work and attached evidence.
5. **Resolved** – Reviewer validated remediation; pending control re-check.
6. **Closed** – Control re-validated and task archived.

3.1 State Transition Diagram

```
Open → In Progress → Ready for Review → Resolved → Closed
↓↑↓
Blocked ■■■■■■■■■■■■■ Reopened → In Progress
```

3.2 Transition Rules

- **Open** → **In Progress** occurs when an assignee acknowledges the task or updates effort estimates.
- **In Progress** → **Blocked** requires entering a dependency note and optional escalation target.
- **Blocked** → **In Progress** triggers notification to watchers documenting the unblock reason.
- **In Progress** → **Ready for Review** only permitted if at least one evidence attachment is linked or a justification is recorded.
- **Ready for Review** → **Resolved** requires reviewer approval and recorded review timestamp.
- **Resolved** → **Closed** is automated once control re-validation succeeds.
- **Any non-Closed state** → **Reopened** is allowed when new findings surface; the system records the reopening reason and increments the reopen_count.

4. Assignment and Escalation Mechanics

4.1 Ownership Model

- **Primary Owner:** Each task maintains a assignee_user_id referencing the accountable individual.
- **Secondary Watchers:** watcher_ids provide visibility for stakeholders (compliance, legal, product).
- **Group Assignment:** Tasks can reference a team_id for routing to shared inboxes; individuals claim ownership via "Accept Task" action.
- **SLA Tracking:** Every task inherits a due date based on severity (e.g., Critical = 3 days, High = 7 days, Medium = 14 days, Low = 30 days). SLAs drive reminder cadence.

4.2 Escalation Policy

1. **Automated Reminders:** Notifications sent at 50%, 75%, and 100% of SLA usage.

2. **Escalate to Manager:** When overdue by >24 hours, the task escalates to the assignee’s manager (escalation_user_id). Manager becomes co-owner until resolution.

3. **Program-Level Escalation:** Tasks overdue by >72 hours for Critical severity auto-create a ServiceNow incident or Slack alert to the governance channel.

4. **Manual Escalation:** Compliance officers can manually reassign or escalate tasks using /tasks/:id/reassign endpoint while preserving audit history.

5. Evidence Attachments

5.1 Accepted Evidence Types

- Documents: PDF, DOCX, or TXT containing updated policies, procedures, or analysis.
- Configuration exports: JSON/YAML files capturing system settings.
- Screenshots or recordings: PNG, JPG, MP4 demonstrating remediation steps.
- Automated probe output: Structured payloads ingested from integration probes.

5.2 Verification Workflow

1. Assignee uploads evidence using the Evidence Repository service; entries are linked via evidence_links referencing the task.
2. Evidence metadata tracks uploader, checksum, and storage path in MinIO.
3. Reviewers validate evidence authenticity, optionally requesting additional artifacts.
4. Evidence approval status is logged; rejected items transition the task back to **In Progress**.
5. Accepted evidence is retained for minimum 36 months with version control and audit trails.

6. External Tracker Integrations

6.1 Synchronization Rules

- **Jira:** Tasks can be pushed as Jira issues; status changes synchronize bidirectionally. Closing a Jira issue triggers the platform to move the task to **Ready for Review**.
- **ServiceNow:** High-severity tasks can auto-create incidents. Resolution in ServiceNow updates task status and imports closure notes.
- **Webhook Events:** Generic webhooks support integration with other ticketing systems. Events include task creation, status updates, reassignment, and closure.
- **Conflict Resolution:** Platform status is the source of truth; conflicting updates from external systems prompt manual review with merge suggestions displayed in the UI.

6.2 Field Mapping

Platform Field	Jira Field	ServiceNow Field	Notes
title	summary	short_description	Truncated to 255 characters if needed.
description	description	description	Markdown converted to HTML for ServiceNow.
severity	priority	priority	Severity→priority mapping configured per workspace.
status	status	state	State machine translation table ensures accurate mapping.
assignee_user_id	assignee	assigned_to	Uses directory mapping or email matching.
due_date	duedate	due_date	Maintains SLA visibility across platforms.
evidence_links	attachments	attachments	Evidence metadata referenced via secure URLs.

7. Data Storage in tasks Table

7.1 Core Columns

Column	Type	Description
id	UUID (PK)	Unique task identifier.
title	Text	Short summary of remediation action.
description	Text	Detailed context, control references, and remediation steps.
control_id	UUID (FK)	Links to the affected control.
alert_id	UUID (FK, nullable)	Populated when derived from alert.
severity	ENUM (critical,high,medium,low)	Drives SLA and reporting metrics.

Column	Type	Description
status	ENUM (open,in_progress,blocked,review,resolved,closed,reopened)	Lifecycle state (UI maps review ↔ Ready for Review).
assignee_user_id	UUID (FK)	Current owner.
team_id	UUID (FK, nullable)	Routing group.
due_date	Timestamp	SLA deadline.
escalation_user_id	UUID (FK, nullable)	Manager or escalation contact.
reopen_count	Integer	Tracks number of reopen events.
evidence_required	Boolean	Indicates whether evidence is mandatory before review.
metadata	JSONB	Flexible payload for integration identifiers, tags, or automation signals.
created_at / updated_at	Timestamp	Auditable timestamps managed by Prisma.
closed_at	Timestamp (nullable)	Populated when task reaches **Closed**.

7.2 Indexing and Auditing

- Composite index on (status, severity, due_date) for dashboard filters.
- Foreign key indexes on control_id, assignee_user_id, and team_id for efficient joins.
- All status transitions emit audit log entries referencing task_id, actor, previous state, new state, and timestamp.
- Soft deletes avoided; tasks remain immutable with status-based archival for traceability.

8. Control Re-Validation Feedback Loop

1. When a task transitions to **Resolved**, the control monitoring engine schedules re-validation.

2. Automated probes or manual testers verify that remediation steps are effective.

3. Successful re-validation updates associated control status to **Compliant** and logs the verification evidence.

4. If verification fails, the task is automatically reopened with appended failure details and new due date.

5. Upon successful validation, the system updates metrics dashboards, recalculates compliance scores, and closes the task (recording closed_at).

9. Usage Scenarios and Reporting Metrics

9.1 Compliance Officers

Scenario: Reviewing weekly remediation backlog.

- Use severity and SLA dashboards to prioritize overdue tasks.
- Generate reports on task aging, reopen counts, and evidence completeness.
- Monitor cross-framework remediation progress, filtering by control category.
- Export integration status summaries to confirm Jira/ServiceNow parity.

Key Metrics:

- Percentage of tasks resolved within SLA by severity.
- Average time from creation to Ready for Review.
- Evidence attachment ratio (tasks with approved evidence ÷ total tasks).

9.2 Engineers and Responders

Scenario: Actioning a newly assigned high-severity task.

- Receive notification via email/Slack with task context and SLA.
- Review linked controls, alerts, and prior evidence to scope remediation.
- Attach remediation proof (config diffs, deployment logs) before submitting for review.
- Update status to **Ready for Review** and monitor for reviewer feedback.

Key Metrics:

- Mean time to acknowledge (MTTA) and mean time to remediate (MTTR).
- Blocked task count by dependency type (infrastructure, vendor, access).
- Reopen frequency by assignee to identify training needs.

9.3 Auditors

Scenario: Preparing for quarterly compliance audit.

- Access read-only dashboards showing closed tasks mapped to control objectives.
- Download evidence bundles with timestamps, reviewer approvals, and re-validation outcomes.
- Filter tasks by audit finding source to trace remediation effectiveness.
- Confirm traceability from finding → task → evidence → control re-validation.

Key Metrics:

- Audit finding closure rate and time to closure.
- Control re-validation success rate post task completion.
- Evidence integrity score (percentage of evidence with checksum verification).

This document defines the operational blueprint for managing remediation tasks, ensuring that governance gaps are tracked, remediated, and verified with full auditability.

Dashboard and Reporting System

Location: /client/src/features/dashboards, /server/src/modules/reports

TL;DR

The dashboard and reporting system transforms governance telemetry into actionable insights and exports.

React dashboards in client/src/features/dashboards consume APIs backed by server/src/modules/reports to visualize scores, observations, and remediation progress.

This runbook describes the data pipelines, UI components, report generators, and extensibility patterns that keep analytics current and trustworthy.

1. Overview

The dashboard and reporting system consolidates program assessment data, turning ongoing evaluation activities into actionable visuals and exportable reports. This document outlines the source pipelines for the key dashboard feeds, the front-end components that render them, and the relationships between report types, backend services, and data storage. Guidance for extending the system with new widgets or exports is also provided.

2. Data Pipelines Feeding Dashboards

2.1 Scores Pipeline

- **Source Events:** Assessment submissions, rubric scoring events, automated performance checks.
- **Ingestion:** Events captured through the `assessment_service` Kafka topic are normalized in the `score_processor` job.
- **Transformation:** The processor calculates aggregated scores (per site, per program, per evaluator) and writes results into the `assessment_scores` table.

- **Serving Layer:** The GET /api/v1/dashboards/scores endpoint exposes denormalized score summaries with pagination, filters for date range, evaluator, and program cohorts.

2.2 Observations Pipeline

- **Source Events:** Classroom observations, on-site audits, follow-up notes.
- **Ingestion:** Observation forms are submitted via POST /api/v1/observations and queued for enrichment by the observation_enricher worker.
- **Transformation:** Natural-language tagging, sentiment extraction, and categorical labeling are stored in observation_tags. Consolidated observation records live in observation_facts with foreign keys to the original submissions.
- **Serving Layer:** The dashboard reads from GET /api/v1/dashboards/observations, which combines facts and tag metadata to generate summaries by site, observer, and theme.

2.3 Tasks Pipeline

- **Source Events:** Corrective action items, follow-up tasks spawned from observations, scheduled compliance tasks.
- **Ingestion:** Tasks originate in POST /api/v1/tasks and are propagated through the task_dispatcher queue.
- **Transformation:** The task_sync job enriches tasks with SLA deadlines, status rollups, and dependency chains, storing results in task_ledger and task_dependencies tables.
- **Serving Layer:** The GET /api/v1/dashboards/tasks endpoint exposes grouped task metrics (open vs. closed, SLA breaches, responsible owners) and powers the task completion widgets.

3. Front-End Visualization Components

Component	*Responsibility*	*Primary Data Source*
ScoreTrendWidget	Line and bar charts for aggregate scores across cohorts and time ranges.	/api/v1/dashboards/scores
ObservationInsightPanel	Displays sentiment trends, most frequent tags, and qualitative highlights.	/api/v1/dashboards/observations
TaskComplianceMatrix	Matrix view showing status of tasks by owner and due date bucket.	/api/v1/dashboards/tasks

Component	Responsibility	Primary Data Source
ProgramDrilldownModal	Contextual details for selected site/program, aggregating scores, observations, and tasks in a single view.	Combination of dashboard endpoints plus /api/v1/programs/:id
ExportMenu	Allows exporting current views and reports; integrates with report generation endpoints.	/api/v1/reports/*

All widgets use a shared DashboardDataContext for state management, ensuring consistent filters (date range, program, evaluator) across panels.

4. Report Types and Backend Relationships

4.1 Scorecards

- **Purpose:** Provide a snapshot of performance metrics for a single site or program.
- **Endpoint:** POST /api/v1/reports/scorecards initiates generation; GET /api/v1/reports/scorecards/:job_id returns the finished PDF/CSV.
- **Database Tables:** Relies on assessment_scores for quantitative metrics and program_metadata for contextual details.
- **Front-End Integration:** The ScorecardReportModal allows users to configure filters and launch scorecard exports directly from score widgets.

4.2 Gap Analyses

- **Purpose:** Highlight discrepancies between target benchmarks and actual performance across programs or standards.
- **Endpoint:** POST /api/v1/reports/gap-analyses with benchmark parameters; GET /api/v1/reports/gap-analyses/:job_id fetches the resulting report.
- **Database Tables:** Uses benchmark_targets, assessment_scores, and observation_facts to compute variance and qualitative explanations.
- **Front-End Integration:** The GapAnalysisBuilder component surfaces recommended actions based on the generated report and cross-links to task creation.

4.3 Risk Heatmaps

- **Purpose:** Visualize compounded risk by combining compliance gaps, open tasks, and negative observation trends.

- **Endpoint:** POST /api/v1/reports/risk-heatmaps; status retrieval through GET /api/v1/reports/risk-heatmaps/:job_id.
- **Database Tables:** Aggregates data from risk_thresholds, task_ledger, observation_tags, and assessment_scores.
- **Front-End Integration:** The RiskHeatmapView renders the produced heatmap tiles and enables drilldowns into contributing factors.

5. Extensibility Guidance

5.1 Adding New Dashboard Widgets

1. **Define Data Contract:** Expose a new backend endpoint (e.g., /api/v1/dashboards/<resource>) returning normalized JSON with filter metadata compatible with DashboardDataContext.

2. **Extend Context:** Update DashboardDataContext to register the new dataset, ensuring global filters propagate appropriately.

3. **Create Visualization Component:** Build a dedicated widget leveraging shared chart primitives (e.g., useChartTheme, BaseCard). Follow accessibility guidelines (keyboard navigation, ARIA labels for interactive elements).

4. **Register Widget:** Add the widget to the dashboard layout configuration with responsive breakpoints and loading/error states.

5.2 Adding New Report Exports

1. **Backend Pipeline:** Implement a new report generator job under /api/v1/reports/<type> with asynchronous processing and job tracking in report_jobs.

2. **Data Model Alignment:** Introduce or reuse database tables for required metrics; define indexes for heavy aggregation queries.

3. **Front-End Entry Points:** Update ExportMenu and relevant widgets to present the new export option, ensuring the request payload matches the backend contract.

4. **File Formats and Localization:** Support PDF and CSV exports where applicable, using shared templates for branding and locale-aware formatting.

5.3 Accessibility and Performance Standards

- **Accessibility:** All widgets must meet WCAG 2.1 AA; ensure semantic HTML, ARIA annotations for charts (role="img" with descriptive labels), and keyboard-accessible controls. Provide text alternatives for visualizations via summary tables or descriptive captions.

- **Performance:** Dashboard endpoints should respond within 300ms for typical queries; leverage caching (Redis) for repeated filters. Front-end widgets must employ lazy loading and memoization to avoid redundant renders. Large report generation tasks should be queued with progress polling to keep the UI responsive.
- **Monitoring:** Instrument both backend jobs and front-end components with tracing/metrics (OpenTelemetry), and set alerts for SLA breaches on endpoint latency or queue backlogs.

6. Future Enhancements

- Introduce machine learning-driven insights (e.g., predictive risk scoring) as optional overlays on existing widgets.
- Expand the report framework with templating support to allow custom combinations of widgets into consolidated executive summaries.

External Integrations System

Location: /server/src/integrations

TL;DR

This document catalogs Project X's third-party integrations, covering supported connectors, authentication models, data synchronization patterns, and operational runbooks.

For every integration we highlight configuration steps, environment variables, failure-handling playbooks, and how the connector ties into notifications, tasks, and evidence lifecycle.

Use this reference when onboarding new environments, planning releases, or troubleshooting production incidents.

1. System Overview

Project X integrates with external governance, collaboration, and evidence systems to enrich policy automation and audit workflows. Integrations adhere to the following principles:

- **Security-first authentication** with short-lived tokens, scoped permissions, and centralized secret storage.
- **Deterministic synchronization** leveraging idempotent polling jobs or event-driven webhooks to prevent duplicate tasks.
- **Modular adapter architecture** where each connector implements the same interface for credentials, fetch, push, and reconciliation methods.
- **Observability** through structured logging, integration-specific metrics, and alert thresholds for synchronization lag, API failures, and payload validation errors.

2. Integration Inventory

2.1 ServiceNow ITSM

Use Cases

- Import incident, change, and request tickets to seed risk assessments.
- Push remediation tasks from automated controls back into ServiceNow for tracking.
- Sync configuration item (CI) ownership metadata into the asset catalog.

Authentication Model

- OAuth 2.0 client credentials flow with dedicated integration user.
- Token scope limited to incident, change_request, task, and cmdb_ci tables.
- Secrets stored in HashiCorp Vault under integrations/servicenow path with automated rotation every 30 days.

Configuration Steps

1. Create an integration user in ServiceNow with itil and rest_service roles.

2. Register an OAuth API endpoint in ServiceNow, noting the client ID and secret.

3. Populate the following environment variables (per environment) and restart the integration workers:

- SERVICENOW_BASE_URL
- SERVICENOW_OAUTH_CLIENT_ID
- SERVICENOW_OAUTH_CLIENT_SECRET
- SERVICENOW_USER
- SERVICENOW_PASSWORD *(fallback for token bootstrap; rotated every 90 days)*

4. Configure the sync schedule in the Integration Control Plane (ICP) UI to poll incidents every 5 minutes and change requests every 15 minutes.

5. Enable outbound webhooks for task updates (REST message definition) pointing to /api/integrations/servicenow/webhook.

Failure Handling

- Automatic retry with exponential backoff (up to 5 attempts) on HTTP 5xx.
- On 401/403 responses, trigger a token refresh; after two failures, page the on-call via PagerDuty.
- Payload validation errors create an internal alert (Severity Medium) with the rejected record attached as evidence.
- Database reconciliation job runs nightly to detect orphaned ServiceNow tasks and queue re-sync operations.

Integration with Notifications / Tasks / Evidence

- New ServiceNow incidents create high-priority review tasks assigned to risk analysts.
- Task updates (state changes, assignment) propagate to Project X's task board via the shared task service.
- Attachments from ServiceNow are mirrored into the Evidence Library, tagged with source:servicenow and related control IDs.
- Notifications: Slack channel #alerts-servicenow receives summaries of sync anomalies and new P1 incidents.

2.2 Atlassian Jira

Use Cases

- Two-way synchronization of engineering remediation tickets.
- Publishing compliance program epics and linking sub-tasks to controls.
- Importing story points and sprint velocity data for operational analytics.

Authentication Model

- OAuth 2.0 (3LO) with offline access, storing refresh tokens per environment.
- For server/DC deployments, personal access tokens (PAT) supported as a fallback.
- Secrets stored in Vault integrations/jira/{env} with audit logging enabled.

Configuration Steps

1. Register a Jira OAuth app; capture client ID, secret, redirect URI (https://<projectx>/oauth/callback/jira).

2. In Project X admin UI, authorize the workspace; the ICP stores the refresh token automatically.

3. Define project mappings (Project X program → Jira project keys) within the Integration Mapping panel.

4. Configure environment variables:

- JIRA_BASE_URL
- JIRA_OAUTH_CLIENT_ID
- JIRA_OAUTH_CLIENT_SECRET
- JIRA_WEBHOOK_SECRET

5. Set up Jira webhooks for issue created/updated/deleted events pointing to /api/integrations/jira/webhook.

6. For PAT mode, set JIRA_AUTH_MODE=pat and provide JIRA_PAT and JIRA_USER_EMAIL.

Failure Handling

- Queue-based retry for webhook deliveries with dead-letter queue after 10 attempts.
- Detect and reconcile deleted issues via nightly full sync (flag ENABLE_JIRA_DELTA_RECON=true).
- Permission errors surface as actionable alerts in the Integration Health dashboard with guidance to re-run the OAuth flow.
- Rate limit responses (429) trigger adaptive throttling; backlog metrics feed Prometheus alert jira_sync_backlog.

Integration with Notifications / Tasks / Evidence

- Jira issues map to Project X remediation tasks, preserving labels and assignees.
- Sprint completion triggers notifications to compliance program owners summarizing outstanding tasks.
- Attachments and comments relevant to compliance controls are copied to Evidence Library with bi-directional links.
- Slack integration posts to #compliance-delivery on failed syncs or when critical Jira epics move to "Done".

2.3 OneTrust GRC

Use Cases

- Import privacy impact assessments (PIAs) and risk registers.
- Synchronize vendor assessments to maintain third-party risk posture.
- Export control attestations back into OneTrust for audit.

Authentication Model

- API token-based authentication using OneTrust Service Accounts.
- Tokens scoped to modules: Assessments, Risks, Vendors, and Controls.
- Stored in Vault integrations/onetrust with automated expiry checks (tokens valid for 1 year).

Configuration Steps

1. Provision a OneTrust service account and assign necessary module permissions.

2. Generate an API token via OneTrust admin console.

3. Populate environment variables:

- ONETRUST_BASE_URL
- ONETRUST_API_TOKEN
- ONETRUST_REGION *(e.g., us, eu)*
- ONETRUST_TIMEOUT_SECONDS *(defaults to 30)*

4. Configure data domain mappings (PIA → Program, Vendor → Third-Party Catalog) in ICP.

5. Enable scheduled imports (recommended: every 6 hours) and configure deduplication rules (use OneTrust external IDs).

Failure Handling

- 401 responses trigger immediate token rotation workflow; integration pauses while awaiting new credentials.
- Validation errors log to Evidence Library as rejected payloads; operations review weekly.
- Timeout or network failures escalate to InfoSec after three consecutive occurrences.
- Stale data detection: if no successful sync in 24 hours, status light turns red and triggers Slack alert.

Integration with Notifications / Tasks / Evidence

- Imported PIAs create review tasks for privacy officers with due dates driven by PIA risk level.
- Vendor issues generate alerts to vendor management Slack channel and create follow-up questionnaires.
- Evidence snapshots (e.g., PIA PDFs) stored in the Evidence Library, linked to regulatory requirements.
- Completion of remediation actions in Project X exports control attestations to OneTrust via scheduled jobs.

2.4 Slack Workspace

Use Cases

- Deliver real-time alerts for integration health, audit tasks, and policy changes.
- Collect evidence and approvals via Slack actions.
- Support chatbot-style queries into compliance status.

Authentication Model

- Slack OAuth 2.0 (Bot token) with granular scopes: chat:write, commands, users:read, files:write, channels:read.
- Signing secrets stored in Vault integrations/slack.

Configuration Steps

1. Create a Slack app per workspace; enable Event Subscriptions and Interactivity.
2. Set redirect URL to https://<projectx>/oauth/callback/slack and install the app.
3. Configure environment variables:

- SLACK_CLIENT_ID
- SLACK_CLIENT_SECRET
- SLACK_SIGNING_SECRET
- SLACK_BOT_TOKEN
- SLACK_APP_LEVEL_TOKEN *(for Socket Mode, optional)*

4. Define default channels in the Notification Routing table (e.g., alerts-servicenow, compliance-delivery).
5. Enable command handlers (/projectx-task, /projectx-evidence) in the admin console.

Failure Handling

- Event delivery failures log to the Notification Service; retries scheduled with jitter (max 6 attempts).
- Expired tokens prompt admin notification with link to reinstall the app.
- Slack rate limits handled via Retry-After header; backlog metrics exported to Prometheus.
- File upload errors generate fallback email notifications to ensure evidence collection continuity.

Integration with Notifications / Tasks / Evidence

- All integration alerts funnel through Slack channels for immediate visibility.
- Task assignments trigger direct messages to assignees with contextual deep links.
- Evidence requests allow users to upload files via Slack modal; files stored in Evidence Library with metadata referencing the originating control.
- Slack actions (approve/deny) update task status and trigger downstream notifications.

2.5 Email + Webhook Bridge

Use Cases

- Support partners without native API support through templated emails or generic webhooks.
- Provide fallback notifications when Slack is unavailable.

Authentication Model

- SMTP credentials stored securely; webhook endpoints protected via HMAC signatures.

Configuration Steps

1. Configure SMTP relay (e.g., SendGrid) and set environment variables:

- SMTP_HOST
- SMTP_PORT
- SMTP_USERNAME
- SMTP_PASSWORD
- SMTP_FROM_ADDRESS

2. For webhooks, define shared secret GENERIC_WEBHOOK_SECRET and register endpoints with trusted partners.

3. Map inbound email addresses to integration adapters (e.g., servicenow-ingest@projectx.io).

4. Enable parsing templates for supported payloads; configure in ICP via Email Bridge section.

Failure Handling

- Bounce or delivery failures raise alerts in Notification Service and attempt resend after 10 minutes.
- HMAC signature mismatches reject payloads and notify security team.
- Email parsing errors stored in quarantine queue for manual review.

Integration with Notifications / Tasks / Evidence

- Email-based submissions create evidence records tagged with source email and attachments.
- Webhook bridge can create tasks or comments based on payload type.
- Failure alerts propagate to Slack and email distribution lists.

2.6 Evidence Repository Providers

Use Cases

- Connect to third-party storage (AWS S3, Google Drive, SharePoint) for evidence ingestion and archival.

Authentication Model

- S3: IAM role assumption using AWS STS and service-linked roles.
- Google Drive: Service Account with delegated domain-wide authority.
- SharePoint: Azure AD app with Files.ReadWrite.All permission and certificate-based auth.

Configuration Steps

- **AWS S3**

1. Create IAM role ProjectXEvidenceRole with trust relationship for Project X integration account.
2. Set environment variables: EVIDENCE_S3_BUCKET, AWS_ROLE_ARN, AWS_EXTERNAL_ID.
3. Configure bucket policies for least privilege (GetObject, PutObject, ListBucket).

- **Google Drive**

1. Create service account and upload JSON key to Vault; set GOOGLE_APPLICATION_CREDENTIALS path.
2. Enable Drive API and grant access to evidence folders.
3. Set EVIDENCE_GOOGLE_DRIVE_FOLDER_ID.

- **SharePoint**

1. Register Azure AD app; upload certificate; capture CLIENT_ID, TENANT_ID, CERT_PATH.
2. Configure site and document library IDs via SHAREPOINT_SITE_ID, SHAREPOINT_LIBRARY_ID.
3. Store certificate password in Vault and reference via SHAREPOINT_CERT_PASSWORD.

Failure Handling

- Storage quota alerts escalate to operations and pause ingestion for affected provider.
- Permission errors generate actionable tasks to update access policies.
- Upload/download retries (3 attempts) before raising error events.

Integration with Notifications / Tasks / Evidence

- Evidence ingestion pipelines normalize metadata and link to controls.
- Notifications inform control owners when new evidence arrives or if ingestion fails.
- Tasks auto-generate for expiring evidence requiring renewal.

3. Cross-Cutting Patterns

3.1 Authentication Models

- **Central Secret Management**: All credentials stored in Vault; integration workers retrieve secrets using short-lived Vault tokens derived from Kubernetes service accounts.
- **Rotation Schedules**: OAuth tokens refreshed automatically; static tokens rotated at least every 90 days. Rotation events logged to Audit Trail.
- **Access Governance**: Integration service accounts managed via least privilege roles; access requests require security approval.

3.2 Synchronization and Scheduling

- **Polling Jobs**: Managed via Quartz scheduler; each job configured with jitter to avoid thundering herd and supports pause/resume.
- **Webhooks**: Preferred for near-real-time updates. Incoming payloads validated via signatures and schema enforcement.
- **Delta Tracking**: Connectors maintain last_seen_at cursors stored in PostgreSQL to ensure idempotent processing.
- **Reconciliation Jobs**: Nightly jobs compare external state vs. Project X; discrepancies create tasks or automatically heal.

3.3 Error Detection and Failure Handling

- **Observability Stack**: Metrics exported to Prometheus (integration_status, sync_latency_seconds, webhook_failures_total), logs shipped to ELK with integration tags.
- **Alerting**: Alertmanager routes integration incidents to Slack and PagerDuty based on severity.
- **Circuit Breakers**: After repeated failures, connectors enter degraded mode, pausing outbound writes while allowing inbound reads.
- **Runbook Links**: Each connector references runbook IDs (see Section 6) surfaced in alerts for quick remediation.

3.4 Notifications, Tasks, and Evidence Mapping

- **Notification Router**: Integration events publish to the Notification Service, which fans out to Slack, email, or in-app alerts based on routing rules.

- **Task Orchestrator**: External tickets and assessments map to internal task objects; status updates propagate bi-directionally.
- **Evidence Library**: All external files ingested undergo virus scanning, metadata extraction, and classification before being linked to controls and audit trails.
- **Audit Logging**: Every external interaction (pull/push) logs correlation IDs enabling traceability across notifications, tasks, and evidence records.

4. Environment Configuration Matrix

Environment	*Vault Path Prefix*	*Sync Frequency Overrides*	*Notification Channels*	*Additional Notes*
Local	integrations/dev	Reduced cadence (ServiceNow 15m, Jira 30m)	Developer DM, test Slack workspace	Use sandbox instances; mock evidence storage.
Staging	integrations/stg	Match production minus high-risk jobs (OneTrust 12h)	#stg-integrations, PagerDuty test service	Run smoke tests on deploy; enable verbose logging.
Production	integrations/prod	Full cadence (ServiceNow 5m, Jira 5m, OneTrust 6h)	#alerts-*, PagerDuty primary	Enforce change window approvals; enable circuit breakers.

Environment variable secrets are injected via Kubernetes secrets referencing Vault. Non-secret configuration (e.g., polling interval) managed via ConfigMaps and Feature Flags Service.

5. Testing and Rollout Strategy

1. **Connector Unit Tests**: Mock external APIs using WireMock; validate authentication handlers, payload schemas, and retry logic.

2. **Integration Sandbox Testing**: Execute end-to-end flows against vendor sandboxes before enabling in staging. Document expected API limits and edge cases.

3. **Contract Testing**: Maintain JSON Schema contracts for webhook payloads; run during CI to prevent breaking changes.

4. **Staging Verification**:

- Run smoke job to import/export sample records.
- Validate notifications (Slack, email) and task creation.
- Verify evidence attachments stored correctly and accessible.

5. **Progressive Rollout**:

- Enable feature flag integration.<name>.enabled for pilot teams.
- Monitor metrics (sync_latency_seconds, webhook_failures_total) for 24 hours.
- Expand to full org after success criteria met.

6. **User Acceptance**: Collect feedback from compliance owners and adjust mappings before production go-live.

7. **Post-Deployment Review**: After rollout, review logs, metrics, and runbook actions; capture lessons learned.

6. Operational Runbooks

6.1 Daily Checks

- Review Integration Health dashboard for red/yellow indicators.
- Validate that the most recent sync timestamp for each connector is < 2× the scheduled interval.
- Spot-check tasks/evidence records created in the last 24 hours for accuracy.
- Ensure notification queues are draining (no backlog > 100 messages).

6.2 Incident Response Escalation

1. Assess severity based on impacted connectors and data criticality.

2. Notify on-call via PagerDuty; provide correlation IDs and recent log excerpts.

3. Follow connector-specific runbooks (ServiceNow RB-002, Jira RB-003, OneTrust RB-004, Slack RB-005).

4. Communicate status updates in #incident-bridge; document timeline for post-incident review.

5. After resolution, trigger reconciliation job and monitor for regression.

6.3 Change Management Checklist

- Submit change request including scope, environments, and rollback plan.
- Obtain approvals from Security, Compliance, and Platform leads.
- Schedule deployment during approved window; announce to stakeholders.

- Execute configuration changes using infrastructure-as-code (Terraform/Helm) for traceability.
- Post-change validation: run smoke tests, verify notifications/tasks, update runbook if required.
- Archive change artifacts in Evidence Library tagged change-management.
