ChatGPT

Executive Summary

Enterprises face ever-evolving cyber threats and need more than periodic training to stay resilient. A **centralized resilience and awareness platform** unifies disparate data (training completions, phishing simulations, incident logs, tabletop results, etc.) into a system of record. By correlating signals across **games, LMS, incident systems, and communication logs**, the platform provides end-to-end visibility into security posture [1] [2]. Unlike traditional LMS or training-only tools that track only compliance (e.g. completion rates and click-throughs), this platform measures behavioral change and risk reduction. It goes *beyond "did they finish the course?"* to metrics that indicate whether employees spot, report, and respond to threats [2]. In short, it shifts from vanity KPIs to human-risk metrics that inform decision-makers and executives about actual cyber-resilience [2] [3].

The platform serves as a **system of record** for security-awareness data, avoiding siloed tools and giving leadership a single pane of glass. As Swimlane notes, **"without a system of record, each part of security operates in silos, often creating incomplete or incorrect data"** [1]. By acting as the authoritative control, configuration, and analytics plane, it ensures consistent policies and scoring across all exercises (games, simulations, training) and all tenants. In doing so, it supports a continuous cyber-resilience score for each organization, reflecting investments in security awareness and actual outcomes against threats.

This approach differs fundamentally from standard LMS or standalone training platforms. Those platforms focus on delivering courses or phishing quizzes, but do not integrate with live incidents, communications, or custom simulations. They don't compute a unified resilience metric or enforce organizational policies. Our platform, by contrast, provides **configurable threat scenarios**, user/team management, automated data ingestion from any source, and cross-tenant analytics. It effectively **operates as the "brain"** of an enterprise's human-security program, orchestrating how simulations run, collecting all results, and computing metrics – whereas games and training modules are treated as **data producers/consumers** only, not standalone systems.

Platform Vision and Product Scope

The platform is designed as a unified **Control Plane** and **Data/Analytics Plane** for enterprise security awareness. As the control plane, it handles tenant onboarding, user and role management, policy configuration, and orchestration of communications and simulations. It encompasses all tools needed to manage and operate a multi-tenant environment [4]. For example, it defines which threats (phishing, ransomware, social engineering, etc.) each organization focuses on, sets compliance training requirements, and issues communication tokens for live exercises. Internally, it embodies core services: tenant provisioning, identity management, configuration management, and logging.

As the **Configuration Plane**, the platform stores and distributes settings and content. Administrators use the UI to specify threat scenarios, game parameters, training content, and communication rules for each period or campaign. These configurations are exposed via APIs so that connected games and simulations

can retrieve them but cannot alter them. This keeps game environments decoupled: they evolve independently as *clients* that consume the platform's data, but cannot change platform state.

As the **Data and Analytics Plane**, the platform ingests events from all sources (games, LMS, incident tools, tabletop exercise results, etc.), normalizes and enriches them, and computes metrics and scores. It persistently stores raw events (in MongoDB) and aggregates results in a relational database. A real-time or near-real-time analytics layer continuously updates each organization's **Resilience Score** based on behaviors (e.g. reporting rate, response time, completion of controls) [2] [3]. Dashboards then present both high-level trends and drill-down detail per team, user, or exercise. Importantly, **all analytics logic lives in the platform** – games merely emit data and retrieve scoring information via APIs.

Within this ecosystem, **games and simulations** play a specialized role: they are content engines that generate user events and context data. The platform distributes configuration (scenarios, game rules, communication sessions) to them, and in turn they **produce data** (player responses, scores, telemetry). By design, games *never* write analytics or scoring data directly; they simply call ingest APIs. This ensures platform stability and consistent data handling, even as game vendors update their environments. In summary, the platform is the stable, multi-tenant SaaS backbone, while games and simulations are pluggable components that consume the platform's config and feed it data.

Target Users and Personas

- **Enterprise Security Administrators:** (e.g. CISOs, SOC managers) They onboard the organization, set up security teams and roles, configure threat scenarios and policies (e.g. phishing focus, compliance training requirements), and monitor overall security culture health. They need tools for managing user access, defining campaigns, and enforcing compliance rules.

- **Security & Awareness Program Owners:** These are typically security awareness coordinators or HR compliance officers. They design training curricula, schedule phishing simulations, and review team-level metrics. They use the platform to track campaign progress, analyze user behavior (click rates, reporting) and identify areas needing more training.

- **Compliance and Risk Teams:** These users require reports and dashboards that tie security awareness activities to risk reduction. They look at aggregated metrics and resilience scores, compliance audit trails, and may drill into specific incidents. They ensure that the program meets regulatory standards and organizational policies.

- **Executive and Leadership Viewers:** Senior management (CIO, COO, Board) want summarized insights: "Is our human firewall strong?" They see high-level resilience scores, trend graphs, and risk indicators. For example, "X% of users can identify simulated phishing" or "Time to report incidents trending down." The platform provides executive dashboards with context for business decisions, not raw training numbers.

- **Internal Platform Operators (Dev/Ops):** The IT team running this SaaS needs operational visibility: health, performance, logs, user support. They manage the multi-tenant infrastructure, CI/CD pipelines, and ensure uptime. They use separate admin tools or dashboards to monitor resource usage across tenants, troubleshoot issues, and perform compliance audits on the platform itself.

Core Use Cases and User Flows

- **Organization Onboarding:** A new tenant (organization) signs up. An admin user (or invited identity) is created in the identity provider and linked to a new tenant record. The platform provisions default settings: initial resilience score, default threat areas, and basic role templates. This may involve creating a directory in the identity provider or registering the tenant domain. The admin then configures the organizational profile (name, regions, time zone). Onboarding may use an invitation link or self-service registration flow.

- **User and Role Management:** Admins can add users by email/domain, assign them to teams or departments, and give them roles (e.g. “Phishing Trainer”, “Team Member”, “Viewer”). The system integrates with an external IdP for authentication, but maintains mapping of each user’s tenant and roles. Users authenticate via the IdP, and the platform enforces RBAC, so e.g. a regular user only accesses their own data while an admin sees all team stats.

- **Threat and Policy Configuration:** A program owner configures which security threats and campaigns to focus on (e.g. quarterly phishing, social engineering, ransomware scenarios). They can set parameters like campaign frequency, difficulty levels, or policy compliance rules (e.g. mandatory yearly training modules). These configurations are saved in the platform’s database and versioned by period. The platform pushes relevant settings to games/simulations via APIs or publish/subscribe: for example, a phishing game might retrieve the latest email templates and target user lists.

- **Game/Simulation Registration:** Third-party game or simulation engines register with the platform through an administrative interface or API. The platform issues each game a client identity and API key or token. The game defines what type of events it will send (e.g. “phishing-click”, “password-reset-challenge”). The platform stores metadata about the game (vendor, version, capabilities) so it can route events correctly.

- **Ingesting Game Events:** During a simulation, the game’s client SDK or backend calls the platform’s ingestion API for each relevant event (e.g. user opened simulated phish, user reported it, completed a quiz). Each HTTP request includes the tenant ID (or is in context of tenant’s API key), and the payload. The API layer authenticates the call, validates schema, then stores the raw event in MongoDB with tenant tag, and enqueues it for processing.

- **Ingesting External Data:** The platform also ingests LMS completions, SOC incident logs, and tabletop exercise results. This might be via scheduled pulls (e.g. LMS CSV import), API connectors, or webhooks. For example, an LMS system could push “user X completed training Y” events to the platform API. All incoming data is tagged by tenant and type, validated, and stored in raw ingestion store.

- **Communication Session Creation:** When a game or simulation requires a live collaboration session (e.g. a ransomware simulation with a mock admin call or a chat channel), it requests this from the platform via a dedicated API. The platform uses Azure Communication Services (ACS) or Microsoft Teams APIs to create the room/call/chat. It creates ACS identities for participants and issues them short-lived access tokens [5]. The platform enforces policies: only authorized users (by role/team)

can join, and the session is linked to the tenant. The game receives a session descriptor and tokens to connect client-side. The platform logs metadata (who joined, when, usage).

- **Viewing Trends and Resilience Scores:** Dashboards in the admin portal show aggregated metrics per organization: e.g. overall phishing failure rate, average report time, training completion percentage, and the computed Resilience Score. Users can filter by team or time period. A drill-down view lets admins inspect individual user or session stats. For example, an admin might click into “Phishing Simulation Q4” and see which users fell for which templates and how quickly people reported attacks. The Resilience Score is continuously updated from incoming data to reflect current posture.

Functional Requirements

- **Organization/Tenant Management:** Support creation of tenant records with unique IDs. Enforce tenant isolation (each dataset is partitioned). Allow custom branding or settings per tenant. Provide APIs for tenant lifecycle (create, update, deactivate).

- **User, Team, and Role Management:** Multi-user support per tenant. Assign users to teams/departments. Role-Based Access Control (RBAC) within tenant (e.g. Admin, Trainer, Viewer). Roles determine UI permissions and API privileges (e.g. only admins can modify config). Integration with external IdP for SSO; map external groups/claims to platform roles.

- **Threat and Policy Configuration:** UI forms to define active threat categories (phishing, social engineering, etc.), periodic goals, and compliance policies. Ability to upload or author training content and phishing templates. Schedule campaigns and set difficulty. Save historical configurations for auditing.

- **Game/Simulation Registration:** A mechanism to register and manage external simulation engines. Store metadata like game ID, version, and endpoint. Issue each game a client credential. Allow administrators to enable/disable games per tenant. Validate game requests against registered sources.

- **Data Ingestion APIs:** RESTful endpoints (secured) for games and external systems to POST event data. Support batch and streaming ingestion. Enforce JSON schemas per event type. Provide webhooks or connectors for third-party services (e.g. LMS, SIEM). Guarantee idempotency or deduplication for events.

- **Analytics and Scoring Features:** Services to process raw events and compute metrics. Define metric formulas (e.g. reporting rate = reports/(reports+clicks)). Combine metrics into a weighted Resilience Score. Allow scheduling of full recalculations or incremental updates. Support exporting data for further analysis.

- **Dashboards and Reporting:** Interactive UI for charts (line graphs, bar charts, heatmaps). Tables of top-risk users or repeating offenders. Executive summary views vs granular drill-down. PDF/CSV export of reports. Alerting: email or Teams notifications when metrics cross thresholds.

- **Communication Orchestration:** APIs to create/join ACS chat rooms, calls, and/or Teams meetings. Manage ACS identities and tokens on behalf of tenants [5]. Track which participants (users or roles) are in each session. Enforce policies (e.g. approval needed to start a call). Log session metadata (timestamps, participants) in audit store.

- **Audit and Governance:** Maintain an immutable audit log of all critical actions: config changes, user role changes, data ingestion events, API calls, etc. Include who did what and when. Support export for compliance. Store e-mails or notifications of simulated incident outcomes. Integrate with logging/monitoring systems (Azure Monitor, Application Insights) for platform health and security events.

Non-Functional Requirements

- **Tenant Isolation:** Enforce strict logical isolation so one organization's data and users cannot affect another's. Use mechanisms such as schema/row filters or separate data partitions to separate data [6]. Test for permission boundaries. Apply RBAC globally so tenants cannot escalate privileges.

- **Scalability:** Design for horizontal scale. Use cloud-managed services (AKS/App Service, Azure SQL, Cosmos DB/MongoDB, Service Bus) that auto-scale. Ensure ingestion pipelines and API tiers can handle surges (phishing campaigns). Employ autoscaling and elastic pools to handle multi-tenant load spikes [7] [8].

- **Performance:** Aim for low-latency APIs (<200ms typical). Ingestion endpoints should acknowledge rapidly, with processing deferred to background workers. Analytics updates should be near-real-time (for instance, dashboards update within minutes). If shared infrastructure is used, carefully size messaging and DB resources to avoid "noisy neighbor" performance degradation [6] [9].

- **Availability/Reliability:** Target at least 99.9% uptime. Deploy critical services (API, DB, cache) across multiple zones/regions. Use Azure resilient services (cosmosDB, SQL with read replicas). For new releases, use blue-green or canary deployments; roll out changes progressively to minimize widespread impact [10]. Include fallback paths (e.g. degrade analytics if some component is down). Monitor SLAs and fail over seamlessly.

- **Security and Compliance:** Use HTTPS/TLS for all communications. Apply OWASP API security best practices (validate inputs, rate-limit, auth checks). Protect data at rest and in transit: encrypt sensitive fields, and use key management (e.g. Azure Key Vault) for secrets. Implement DDoS protection and Web Application Firewall on API gateway. Leverage Azure compliance certifications (ISO 27001, SOC2) to meet enterprise requirements. Conduct regular penetration tests and code reviews.

- **Observability:** Instrument the platform for full observability. Integrate logging, metrics, and tracing (Azure Monitor, Application Insights). Track user activity, API usage, system metrics (CPU, memory, queue depths). Set up alerts for anomalies (e.g. sudden spike in errors or ingestion latency). Provide per-tenant usage dashboards (calls per day, storage used). Ensure logs contain tenant context for debugging.

High-Level Architecture

The high-level architecture consists of:

- **Platform Services:** Central microservices or APIs built on ASP.NET Core. Key services include: *Identity/Authentication Service* (token validation, tenant claims), *Configuration Service* (manages policies and game configs), *Ingestion Service* (HTTP endpoints for events), *Analytics Engine* (computes metrics and scores), *Communication Orchestrator* (ACS/Teams integration), and *Reporting/Dashboard Service*.

- **Games and Simulation Clients:** External applications (phishing games, attack simulators, LMS systems) running independently. They authenticate with the platform's ingestion API to submit events, and with the communication service to join sessions (via ACS tokens). They pull configuration from the platform via API.

- **External Data Sources:** LMS/training platforms, SIEM/Incident systems, and tabletop exercise tools. These either push data to the platform or are polled periodically. They may integrate via secure API endpoints or connectors (e.g. built-in LMS adapters).

- **Identity Provider:** An external OAuth/OIDC provider (e.g. Azure AD B2C, Kinde, Auth0). All user authentication and initial authorization occur here. Upon login, the platform issues its own JWT (including tenant ID, roles) to users. Service principals or client credentials are used by games/systems.

- **Communication Services:** Azure Communication Services (ACS) for real-time chat/call, and/or Microsoft Teams via Graph API. The platform uses ACS SDK/server APIs to manage identities and tokens [5]. For Teams, the Graph API is used to create channels, chats, or meetings and to retrieve call records [11] [12].

- **Databases and Storage:** A **relational database** (e.g. Azure SQL or PostgreSQL) stores core platform data: tenants, users, roles, configurations, aggregated analytics tables, and projections for reporting. A **document/event database** (MongoDB) holds raw ingestion data (game events, LMS logs, etc.). Both support partitioning by tenant.

- **Background Processing:** Event-driven workers (e.g. Azure Functions or hosted services) consume from queues/topics. For example, after an event is written to MongoDB or placed on Azure Service Bus, a worker normalizes it and updates the relational read models or computes incremental metrics.

- **Messaging and Queues:** Use Azure Service Bus or Event Hub to decouple ingestion from processing. Ingested events are published to a message bus, enabling asynchronous processing and retrying of analytics. This also allows scaling of consumers independently.

- **Monitoring/Logging:** Central logging using Azure Monitor/Log Analytics. Application logs, performance metrics, and alerts are aggregated. Each log entry includes tenant context to correlate issues.

This architecture emphasizes **loose coupling**. All external inputs (games, LMS, comms) interface only through well-defined platform APIs. Communication with ACS/Teams is encapsulated in the Communication Orchestrator service. The relational database serves as the **single source of truth** for configuration and aggregated analytics. MongoDB is the append-only event log store.

Platform vs Game Boundary

We maintain a strict separation of concerns between the platform and games/simulations. The platform **owns all configuration and analytics**. Games consume configuration (threat settings, user assignments) via read-only APIs and produce events via ingestion APIs. Importantly:

- **Stable API Contracts:** The platform exposes versioned RESTful APIs for ingestion, config retrieval, and communication. These APIs form a clear contract. Games must adhere to these APIs; any changes (e.g. new fields) must be versioned so as not to break existing clients. Semantic versioning should be used, and backward compatibility maintained when possible.

- **No Direct Writes to Platform Data:** Games never write directly into the platform's databases. All data must go through the official APIs. This ensures consistent validation, authorization, and business logic. It also means game developers are insulated from platform schema changes.

- **Tenancy Enforcement at Platform:** Games request operations with a tenant context. The platform authenticates these requests (via tokens or API keys) and enforces that a game can only submit events for tenants it is registered with. Tenant identification is passed explicitly (or via subdomain/endpoint) so the platform can segregate data.

- **Communication Sessions:** Games do not create or manage communication rooms on their own. Instead, they call the platform's Communication API to request a session. The platform then provisions it via Azure Communication Services or Teams. Games are given ephemeral tokens only for connecting to that session. This ensures consistent identity management and audit logging [5].

In sum, the platform *controls the flow of information*, and games are sandboxed clients. This decoupling lets games evolve (e.g. new simulation logic, updated scoring methods) without destabilizing the core platform, and vice versa.

Detailed Backend Architecture

- **API Layer:** ASP.NET Core Web API (or minimal APIs) expose all endpoints. Controllers are organized by function (Tenant, User, Config, Ingest, Analytics, Comm). Use API routing that includes tenant context (e.g. via header or URL). Employ API Management as a gateway to handle throttling, caching, and to manage subscriptions for multi-tenant auth [13].

- **Domain and Business Logic:** Implement a layered or onion architecture (Clean Architecture) for maintainability [14]. The **core domain layer** contains entities (e.g. Tenant, User, ThreatConfig, Event) and business rules. Surrounding layers implement use cases and interfaces. For example, an

`IEventRepository` interface in the domain layer is implemented by an EF Core or Mongo repository in the infrastructure layer. This keeps business logic independent of frameworks.

- **Ingestion and Normalization:** The ingestion service validates incoming payloads (using Data Annotations or a library like FluentValidation) and tags them with tenant and event type. Raw events are stored in MongoDB using a flexible schema. Each event type can have its own collection. After storing, the API enqueues a message on Azure Service Bus for processing.

- **Analytics and Scoring Services:** Worker services (could be Azure Functions, Azure WebJobs, or hosted background services) listen to the queues. They read raw events, then normalize and project them into analytics. For example, a "PhishingClick" event might increment a counter in a `UserPhishingStats` table in SQL. Workers apply time windows (daily, monthly) to compute trending metrics. A separate *Scoring Engine* aggregates metrics into a composite Resilience Score per tenant. Scoring logic is versioned – for instance, a new scoring model (v2.0) can be deployed and applied to new data while preserving past scores for comparison.

- **Communication Orchestration Service:** A dedicated service handles integration with ACS and Teams. When a session is requested, it invokes ACS SDK to create or retrieve communication identities and access tokens [5]. For Teams, it calls Microsoft Graph APIs to create meetings/channels or retrieve meeting info [11]. This service also enforces policies (e.g. only allow calls of certain types) and writes metadata to the database. It uses secrets (ACS connection string, Graph credentials) stored in Key Vault.

- **Background Workers:** Many tasks run asynchronously: event normalization, compliance checks, report generation, email notifications, and housekeeping (e.g. expiring old sessions). These run as background services within the ASP.NET host or separate worker apps. They poll message queues or trigger on timers.

- **Authorization/Tenant Enforcement:** Global middleware inspects each request's token. It extracts the tenant ID and user roles (claims) and sets a request context. All data access layers use this context to filter queries (e.g. adding `WHERE TenantId = X` on every data query). This ensures strict isolation at the code level. Even if shared DB or shared services are used, the application code never returns data belonging to another tenant.

- **Claims and Identity Handling:** Upon login, the external IdP issues a JWT with an identifier (and possibly tenant membership). The platform's authentication service creates its own JWT including `tenantId` and `roles` claims [13]. Each API and service trusts this token for authorization decisions. For service-to-service calls (games, LMS), use client credentials: they obtain tokens via the IdP or an internal client registry and call APIs as a machine identity, with permissions limited to their functions.

Frontend Architecture

- **Admin & Configuration Portal:** A single-page application (SPA) built with React and TypeScript. It uses component libraries (e.g. Material-UI or Ant Design) for consistency. State management can use

Redux or React Context to hold user/session info and selected tenant. The portal includes screens for user/team management, configuration of threats/policies, and campaign scheduling.

- **Analytics & Reporting UI:** Dashboards with charts (e.g. Chart.js or D3.js) show trends. Implement dynamic, filterable grids (e.g. using ag-Grid). Users can toggle between executive view (high-level resilience score and charts) and detailed view (per-team, per-user stats). Reports can be generated on demand (via the API) and displayed or downloaded.

- **Communication UI Integration:** Where relevant, embed components for live chat or calls. For ACS, the client app (React) would use the ACS Web SDK, receiving tokens from the platform, to join chats or calls. For Teams, one can use the Teams client or deep links issued by the platform to join a meeting. The UI includes real-time indicators of session status.

- **State Management & Routing:** Use React Router for multi-page navigation. Components are wrapped to enforce RBAC: e.g., an AdminRoute component checks the user's `role` from context/claims before rendering the page. Unauthorized UI elements (buttons, tabs) are hidden based on roles from the JWT.

- **API Integration:** All calls to the backend APIs attach the bearer token. Use Axios or Fetch with global interceptors for auth. Implement retry logic for transient failures. Data fetching is centralized (e.g. a custom hook or service) to allow caching. The UI handles pagination and filtering by passing query parameters to the API and displaying the results.

Data Model

- **Organization/Tenant:** Includes TenantID, name, subscription details, configuration defaults, current Resilience Score. Possibly a `ResilienceSettings` sub-entity for weightings of metrics.

- **User, Membership, Role:** Users have UserID, name, email, tenantId, and roles (Admin, Trainer, Member). Membership links users to teams or departments. Roles determine UI and API permissions. A separate `UserStats` table may store per-user metrics (e.g. report rate, click rate).

- **Teams/Groups:** Logical grouping within a tenant (e.g. departments). Used for managing targeted campaigns and aggregating scores by team.

- **Games and Simulation Sources:** Entries for each registered game or data source, including sourceId, name, type, version, and endpoint info. Used to validate incoming data and route it.

- **Game Sessions and Player Sessions:** Entities to record individual runs of simulations (SessionID, GameID, start/end time, participants). PlayerSession records per-user activity within a game session (e.g. score, completion status).

- **Policies and Threat Configurations:** Entities for storing current configurations: e.g. `ThreatFocus` (phishing: true/false), `TrainingModule` (moduleID, dueDate), `SimulationConfig` (phishing templates, difficulty). These drive what scenarios run.

- **Communication Sessions:** Records of each created call/chat. Contains a SessionID, type (chat or call), tenantId, initiating user, start/end times, and link to ACS/Teams resources. Also stores audit info (who joined).

- **Raw Events:** In MongoDB, store each raw event document with fields: tenantId, sourceId, eventType, timestamp, payload (game-specific fields like {userId, action, result}). Collections are typically per-event-type (e.g. “PhishingEvents”, “ClickEvents”).

- **Normalized Results:** Relational tables where events are aggregated or processed. For example, a `PhishingStatistics` table might store per-user and per-team totals of clicks vs reports. These are updated by workers.

- **Aggregated Metrics and Scores:** Tables for time-series or snapshot metrics: e.g. `DailyResilienceMetrics(tenantId, date, score, components…)`. The composite `ResilienceScore` (numerical) is stored with timestamp.

Multi-Tenant Data Isolation Strategy

We adopt a **shared application code with isolated data layers** approach. Since multiple organizations use the same platform instance, data is partitioned by tenantId in all stores. Options include:

- **Shared Database (Row-Level Security):** Use a single relational DB with tables containing a `TenantID` column. Apply SQL Row-Level Security (RLS) or application filters so every query automatically restricts to the request’s TenantID [15]. This eases cross-tenant reporting (if needed) and scales well for many tenants. RLS provides strong isolation in the database itself.

- **Shard per Tenant (Elastic Pools/Databases):** For very large tenants or compliance needs, we could place each tenant’s data in its own database within an elastic pool [7]. Elastic pools share resources across DBs while mitigating noisy neighbor issues. Azure’s Elastic DB Tools support sharding and managing many databases. This gives strong isolation (data physically separate) at higher cost.

- **Document Store (MongoDB):** Similarly, Mongo can use separate databases or collections per tenant. The simplest is a shared Mongo cluster with each event document tagged by tenantId. For stricter isolation, one could create a database per tenant in Mongo.

Trade-offs: A fully shared database is cost-efficient but requires careful coding to enforce filters [6]. Separate DBs offer maximum isolation (good for compliance) and protect one tenant’s heavy load from others. We can combine: use shared DB with RLS for core app data, and allow optional dedicated DBs for top-tier clients. Azure SQL’s elastic pools help balance cost and performance [7]. Row-level security is recommended for most cases [15].

Overall, the recommended approach is **shared code**, **shared compute**, with **per-tenant data partitioning** via RLS or sharding as needed. All data access is through parameterized queries filtered by tenantId to prevent cross-tenant leaks. Keys/secrets (e.g. connection strings) are not tenant-specific to keep deploy simple, but data segregation is enforced logically.

Ingestion and Event Processing Architecture

- **Game Ingestion APIs:** REST endpoints accept JSON events. The ASP.NET controllers authenticate the caller, then map the JSON into domain objects. Payloads are validated (schematron or JSON schema). Each valid event is immediately stored in the raw events store (MongoDB) with metadata (source, tenant).

- **External System Ingestion:** Similarly, provide connectors or endpoints for LMS or incident feeds. For periodic data (e.g. CSV exports), an API or portal allows bulk upload. The system ingests them into Mongo or staging tables.

- **Validation and Enrichment:** The ingestion layer checks data consistency (timestamps reasonable, required fields present). Enrichment may include looking up user/team IDs from the user directory, adding geo-tags, or correlating with past events. If enrichment fails (unknown user), events go to an error queue for manual review.

- **Raw Event Storage:** All raw inputs are stored immutably in MongoDB. This is the canonical event log. We index by tenant and event type for efficient retrieval.

- **Message Queue:** After storing raw, each event is published to an Azure Service Bus or Event Hub topic. This decouples ingestion from processing and provides buffering. Each event type may go to its own queue or topic subscription.

- **Normalization Pipeline:** Background worker processes read from queues. They parse events and map them into normalized domain events. For example, a “phish_click” event becomes a `UserAction` entity with {userId, actionType, result, timestamp}. The worker then updates analytics models: e.g., increment counters or insert into fact tables.

- **Projection/Update:** Workers may update relational DB tables (analytics read models). For large batches, use bulk operations. If an event is part of a session (game run), it may update or close that session record when game ends.

- **Failure Handling and Retries:** Use poison queues and retry logic. If an event processing fails, log the error and optionally move to a dead-letter queue. Implement idempotency where possible (e.g. include event ID to avoid double-counting on retry). Monitor processing latency and errors via dashboards.

- **Monitoring:** Track the lag between ingestion and processing. Ensure that ingestion rate spikes (e.g. during simulated attack) do not overwhelm the system.

API Design

- **Platform APIs:** Expose standard RESTful JSON APIs using ASP.NET Core. Follow resource-oriented URIs (e.g. `GET /api/organizations/{orgId}/score`). Use API versioning in URL or headers.

Support pagination (`page`, `size`) and filtering (by date, by user) on list endpoints. Use consistent error model (e.g. RFC 7807 Problem Details).

- **Ingestion APIs:** Provide specific endpoints for each event type (e.g. `POST /api/ingest/events/phishing`). Use HTTP POST with JSON body. Allow bulk posts (arrays) for efficiency. Responses should indicate success per item or the entire batch. Return 202 Accepted for asynchronous processing.

- **External Integration APIs:** For LMS, consider an endpoint like `POST /api/ingest/lms/completions` or use WebHooks. Document the payload clearly. Authenticate external systems via OAuth2 client credentials or per-tenant API key.

- **Communication Orchestration APIs:** e.g. `POST /api/comm/sessions` with details (type, participants, policy). `GET /api/comm/sessions/{id}` returns session info (ACS token, endpoints). These APIs call ACS/Graph under the hood.

- **Authentication/Authorization:** Use bearer tokens. Admin APIs require an admin claim. Games use client credentials to obtain tokens. Protect all endpoints with TLS.

- **Pagination/Filtering:** For list endpoints (users, events, sessions), support `?skip`/`?take` or `page`/`limit`. Allow filtering by date, type, or user within the tenant's data.

- **Error Standards:** Return clear HTTP status codes and JSON error objects. For example, 400 for validation errors, 401/403 for auth issues, 429 for rate limits. In batch operations, indicate which items failed and why.

Authentication and Authorization Model

- **External Identity Provider:** Use a dedicated multi-tenant IdP (e.g. Azure AD B2C or equivalent). Each organization's users log in via their corporate identity or a shared directory. The platform only trusts tokens from this IdP.

- **Organization Membership:** The JWT from the IdP includes a claim indicating the tenant or organization. The platform also stores tenant-user mappings; on login it binds the external user to a tenant and role(s). Every request token carries a `tenantId` claim.

- **RBAC and Permissions:** Define roles (Admin, Manager, User, etc.) with associated permissions. Encode roles as claims in the JWT. Use policy-based authorization in ASP.NET Core so that each API or UI action checks for the required role. For example, only users with `role:Admin` can call `POST /config/threats`.

- **Claims Usage:** Beyond `tenantId` and `roles`, include userId and perhaps teamId in tokens if needed. These claims drive filtering in the backend (e.g. a user can only view events for their team). Avoid placing sensitive data in claims; use them only for identity and access.

- **Game Authentication (Service Clients):** Games and simulators authenticate as service principals (client credentials). The platform registers each game as an OAuth application or client, issuing it a secret. The game uses this to get a machine token (with scopes like `ingest.write`). The platform checks that the token's client ID is authorized for the tenant they claim to send data for.

- **External Systems Authentication:** Integrations (LMS, SIEM) are also clients or use similar token flows. Optionally, use mutually authenticated certificates or Azure Managed Identities if the external system resides in Azure.

- **Token Management:** Use short-lived JWTs for user auth (e.g. 1 hour) and refresh via the IdP. For ACS chat/call, issue separate short-lived access tokens via the ACS identity service (as shown [5]). For Teams calls, use Graph's delegated flow with Azure AD tokens.

Analytics and Resilience Scoring Architecture

- **Event Normalization:** Once raw events are ingested, normalization workers translate them into common metrics. For example, a phishing click event and a LMS training completion event are both mapped into fields like `BehaviorType` (e.g. "PhishClick", "TrainingComplete"), with standardized attributes.

- **Metric & Feature Extraction:** Compute features such as *reporting rate*, *average dwell time*, *repetition of failures*, and *training compliance percentage*. Use sliding windows or fixed intervals (daily, weekly). A real-time stream processor (or micro-batch) updates these as new events arrive.

- **Scoring Service Design:** The scoring engine takes multiple metrics (from training, phishing, incident response) and combines them into a single **Resilience Score**. This might be a weighted sum or more complex model. For example, high phishing-reporting rate increases the score, while long dwell time lowers it. The model is configurable per tenant or versioned globally.

- **Data Source Fusion:** Each data source contributes different signals. Phishing sims measure human factor, LMS measures formal training, SOC logs measure real incidents detected, etc. The scoring layer normalizes these (e.g. to 0–100) before combining. The design ensures each source's data flows in through the same pipeline and is stored in relational tables for the score calculation.

- **Versioning of Logic:** Since best practices evolve, scoring logic is versioned. The system may tag each score with its calculation version. Historical scores remain auditable. When a new version is deployed, it may backfill past data or start computing new scores from that point.

- **Output:** The computed score (e.g. 0–100) is stored per organization and date. The API provides both the current score and historical trend. Dashboards can display component breakdowns (e.g. "phishing sub-score = 75") for transparency.

Communication Platform Architecture

- **Azure Communication Services / Teams:** The platform leverages **Azure Communication Services (ACS)** and **Microsoft Teams** for all real-time communication needs. ACS provides SDKs for chat and VoIP, with a server-side identity service. When a communication session is needed, the platform's server creates a new ACS user identity and issues a short-lived ACS access token [5]. It then invokes ACS Chat or Calling APIs so that clients (games) can connect securely with that token.

- **Identity and Token Issuance:** The Communication Orchestrator service manages ACS identity creation. For each participant (player or admin) in a simulation, the platform calls `CommunicationIdentityClient.createUser()` to get a user ID, then `token.issue` to get a token with scopes (chat, voip) [5]. These tokens are sent back to the game client, which uses ACS Web SDK to join chats or calls. For Teams integration, the platform can exchange a Teams user AAD token for an ACS token if needed, using the ACS Teams interoperability APIs [16].

- **Session Creation Flow:** When the game requests a session, the platform either sets up an ACS chat thread/call or a new Teams meeting. For Teams, it uses Microsoft Graph (e.g. `POST /onlineMeetings`) to schedule an ad-hoc meeting or channel. The response contains join URLs. The platform records the meeting ID and associated tenant. For ACS chat, the platform calls `CreateChatThread`, specifying participants' ACS IDs, and gets back a thread ID.

- **Policy Enforcement:** Before provisioning, the platform checks policies (e.g. only admins can start an all-hands call, or certain content is disallowed). It enforces room/participant limits and roles (e.g. only designated users get join tokens). During the session, only users with valid tokens can enter, and the platform's own backend can monitor presence if needed.

- **Metadata, Audit and Retention:** The platform logs metadata for every communication session: who created it, start/end times, participant list, and any relevant attributes (e.g. scenario type). If a Teams meeting, the platform can use the Graph call records API to pull additional data after the meeting [12]. These logs are stored in the audit database for compliance and later analysis (e.g. linking a drop in score to a poorly-handled simulation call). Retention policies comply with enterprise standards; for example, ACS logs can be kept in secure storage for X years.

- **Client Integration for Games:** Client SDKs (JavaScript/Unity) in the games use the ACS or Teams client libraries. They do not manage identities directly. Instead, the game calls the platform to *request* a session and receives back everything needed (ACS tokens or Teams links) to connect. This way, client apps remain thin and the platform controls all communication lifecycles.

Security Design

- **Threat Model:** Protect against unauthorized data access (tenant breakout), data tampering, and service abuse. The strongest isolation measure is application-level enforcement of tenant boundaries [6]. We also assume attackers may attempt API scraping or DDoS. Use Azure DDoS Protection and API throttling.

- **API Security:** Only HTTPS endpoints. Implement OWASP API Security principles: validate input payloads to prevent injection, use strong authentication/authorization on every endpoint, rate-limit suspicious IPs/clients, and log all auth failures. Use ASP.NET Core built-in protection (anti-forgery, built-in identity, etc.) and consider API Gateway (APIM) policies for additional validation [13].

- **Data Protection:** All sensitive data at rest should be encrypted: use Transparent Data Encryption on SQL, and Mongo's encryption. Critical fields (like PII) should be Always Encrypted or encrypted in app. Use Azure Key Vault to store connection strings and secrets [17]. Secrets (e.g. ACS keys, Graph client secrets) rotate regularly and never reach client side.

- **Tenant Isolation Risks:** Prevent cross-tenant data leaks. Every data query must include tenantId. Use row-level security and scoped queries. Also isolate tenant uploads (games should tag data with tenant ID; the platform verifies it matches the caller's tenant). Protect against insecure direct object reference by rejecting requests missing proper context.

- **API Abuse:** Include anomaly detection in logs. E.g. alert if a single game client submits abnormally large data volume. Use API Management or WAF to block SQLi/XSS attempts. Set strict CORS policies (only allow known game domains).

- **Secure External Integration:** LMS or SIEM connectors should authenticate securely (e.g. OAuth client secret or managed identity). Any third-party integration should go through vetted connectors. All external calls (e.g. calling Graph API) occur from the server side.

- **Secrets and Key Management:** Store secrets in Azure Key Vault. Use managed identities for platform components to access Key Vault. Never hard-code secrets. Audit vault access and rotate keys on schedule.

- **Compliance:** Ensure audit logs are tamper-evident. Possibly integrate with Azure Monitor or SIEM for log forwarding. Support GDPR/CCPA features: e.g. ability to delete a tenant's data entirely on request.

Deployment and Infrastructure

- **Azure Reference Architecture:** Leverage Azure PaaS where possible. For compute, consider Azure Kubernetes Service (AKS) or App Service with containers for the microservices. Use Azure SQL Database for relational data and Azure Cosmos DB (Mongo API) for events. Use Azure Service Bus/ Event Hubs for messaging. ACS and Azure AD are first-party services.

- **Environments:** Separate dev, staging, and prod subscriptions. Use infrastructure-as-code (ARM templates or Terraform/Bicep) to define all resources. Deploy identical environments except scaled-size. Use feature flags for gradual roll-out.

- **CI/CD:** GitHub Actions or Azure DevOps for build/deploy. On commit to main, run unit tests, build containers, and push to registry. Then deploy to staging (with migration scripts). After smoke tests, promote to prod with zero-downtime (e.g. rolling update in AKS or swap slots in App Service).

- **Infra as Code:** Define VNet, subnets, AKS clusters, databases, and ACS resources in code. Automate tenant provisioning if necessary (e.g. create directory entry, assign roles). Use Bicep/Terraform modules for repeatable patterns.

- **Migration and Rollout:** Plan for schema migrations carefully. For relational DB, use EF Core migrations or SQL DACPAC with online migrations. Roll out new features behind flags. Communicate changes to game developers in advance when API contracts change.

- **Backup and DR:** Use Azure built-in backups for SQL and Cosmos. Configure geo-replication across regions. Define RPO/RTO targets (e.g. 1 hour RPO, 4h RTO) and test restores.

Extensibility and Roadmap

- **New Game/Simulation Engines:** The platform's APIs are generic, so any new type of simulation can integrate by following the ingestion API contracts. For example, adding a VR security training game or AI-driven phishing bot is a matter of registering it and having it post the correct events. The configuration schema can be extended to support new threat types or exercise formats.

- **New Data Source Connectors:** Build adapters for common enterprise sources: e.g. direct connectors for SAP, SIEM tools, more LMS platforms, or HR systems. Use Azure Logic Apps or Functions for quick integrations (e.g. parse ServiceNow incident data).

- **Advanced Analytics & ML:** Future enhancements may include a machine-learning layer that finds patterns in user behavior. For example, anomaly detection on phishing clicks or personalized training recommendations. The platform could export data to Azure ML for training models (phishing susceptibility prediction, targeted coaching).

- **Executive and Regulatory Reporting:** Offer pre-built reports (PDFs) that satisfy regulatory frameworks (e.g. NIST, GDPR training logs). Build a report designer or integration with Power BI. Over time, support dashboards customized to industry standards.

- **Marketplace/Integration Ecosystem:** In the longer term, the platform could expose an API marketplace for third-party content (phishing templates, training modules, threat intelligence feeds). Other software vendors could plug into it via published API specs, extending functionality without altering core code.

Clear Implementation Notes for Another AI

- **Architecture Rules:** Follow Clean Architecture / Onion. Domain entities (ResilienceScore, Event, User, etc.) are central and independent of frameworks [14]. Use ASP.NET Core Web API for controllers, and keep logic in services. Use CQRS where commands (e.g. ingest events) are separated from queries (e.g. dashboard reads).

- **Layering and Naming:** Codebase should have clear projects/modules: e.g. `Domain`, `Application`, `Infrastructure`, `WebAPI`. Within code, prefix DB context classes (e.g.

`ResilienceDbContext`) and use repository patterns with interfaces. Name endpoints with nouns (e.g. `/api/v1/organizations`, `/api/v1/ingest/events`).

- **Project Structure:** A monorepo or microservices can be used. For microservices, isolate by concern (e.g. one service for Ingestion, one for Analytics). Or use a modular monolith. In either case, logical separation (as above) is key. Use solution folders matching layers.

- **Key Libraries/Frameworks:** Use ASP.NET Core (latest LTS), Entity Framework Core for relational data, MongoDB .NET driver for raw events. Use MediatR for CQRS if desired, AutoMapper for DTO mapping, FluentValidation for input validation. For React, use Redux or Zustand, and component libraries as needed. Use Azure SDKs (Azure.Identity, Azure.Communication) for comms.

- **Coding Standards:** Ensure dependency injection everywhere. Use interfaces for all services. Document all API contracts (e.g. via Swagger OpenAPI). Follow Microsoft guidelines for async code. For date-times, use ISO 8601 UTC consistently. For secrets and config, use Azure Key Vault and configuration providers.

- **Integration Standards:** All external calls (ACS, Graph API) happen in dedicated services. Wrap them in interface abstractions for testability. Handle retries on transient failures. For message passing, ensure idempotent message handlers. Use consistent logging at Info/Warning/Error levels.

- **Testing:** Write unit tests for business logic, and integration tests for API endpoints. Mock external services in tests. Include end-to-end tests for critical flows (ingestion to dashboard).

By following these detailed guidelines and architecture decisions, the platform can be implemented robustly by downstream automation.

Sources: Azure SaaS multitenancy guidance [4] [6] [7], security metrics best practices [2] [3], and Azure Communication Services documentation [5] [11] [12].

---

[1] A System of Record for Security: Everything You Need to Know
https://swimlane.com/blog/security-operations-system-of-record/

[2] Security Awareness Metrics That Matter: Predicting Breach Reduction - Hoxhunt
https://hoxhunt.com/blog/security-awareness-metrics

[3] Top 4 Cyber Resilience Metrics Worth Tracking | Bitsight
https://www.bitsight.com/blog/4-cyber-resilience-metrics-track

[4] Control plane vs. application plane - SaaS Architecture Fundamentals
https://docs.aws.amazon.com/whitepapers/latest/saas-architecture-fundamentals/control-plane-vs.-application-plane.html

[5] [16] Create and manage access tokens for end users - An Azure Communication Services guide | Microsoft Learn
https://learn.microsoft.com/en-us/azure/communication-services/quickstarts/identity/access-tokens

[6] [10] Tenancy Models for a Multitenant Solution - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/considerations/tenancy-models

7 15 17 Multitenancy and Azure SQL Database - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/service/sql-database

8 9 Architectural Approaches for Messaging in Multitenant Solutions - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/approaches/messaging

11 Use the Microsoft Graph API to work with Microsoft Teams - Microsoft Graph v1.0 | Microsoft Learn
https://learn.microsoft.com/en-us/graph/api/resources/teams-api-overview?view=graph-rest-1.0

12 Working with the call records API in Microsoft Graph - Microsoft Graph v1.0 | Microsoft Learn
https://learn.microsoft.com/en-us/graph/api/resources/callrecords-api-overview?view=graph-rest-1.0

13 Use Azure API Management in a Multitenant Solution - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/guide/multitenant/service/api-management

14 Onion Architecture In ASP.NET Core With CQRS - Detailed - codewithmukesh
https://codewithmukesh.com/blog/onion-architecture-in-aspnet-core/