🧭 Documento Técnico Narrado – Proyecto SAFE

1. Enfoque general del desarrollo

SAFE (*Sistema Académico de Formación Empresarial*) se concibe como una plataforma modular orientada a la automatización de procesos de formación y evaluación en talento humano.
Su estructura responde a una arquitectura **en capas** basada en el patrón **Modelo–Vista–Controlador (MVC)**, sustentada en **Django** y **PostgreSQL**, con **HTMX** para el intercambio dinámico entre vistas sin necesidad de frameworks pesados del lado del cliente.

El objetivo técnico es construir un sistema coherente, seguro y escalable que soporte distintos perfiles (Analista de Talento Humano, Colaborador y Supervisor), garantizando trazabilidad y comunicación fluida entre los módulos.
El proyecto se desarrollará en **tres semanas**, bajo una distribución de responsabilidades individuales, donde cada integrante abordará tres casos de uso que conforman un módulo funcional completo.

---

2. Organización del equipo y cronograma de trabajo (versión ampliada y reordenada)

El desarrollo de **SAFE** se articula en torno a una planificación **por módulos funcionales** y no simplemente por fases de desarrollo abstractas. Cada integrante del equipo asume tres casos de uso relacionados que, juntos, conforman un subsistema completo dentro de la arquitectura global.

La organización se sustenta en tres principios: **especialización funcional**, **entregas semanales incrementales**, y **sincronización estructurada** mediante control de versiones.
El trabajo se organiza en **tres semanas**, cada una con un enfoque técnico distinto: *fundamento, expansión e integración*.

Semana 1: Fundamento del sistema — autenticación, usuarios y estructura base

Durante la primera semana, el equipo consolida la **infraestructura técnica** del proyecto y sienta las bases del flujo de acceso y control.
En esta etapa se crean las aplicaciones base de Django (`accounts`, `core`, `courses`) y se establecen las relaciones fundamentales de la base de datos.
Es la fase en la que se alinean el *backend*, la estructura de carpetas y la conexión con PostgreSQL, con la finalidad de tener un entorno de desarrollo funcional antes de avanzar a módulos dependientes.

Objetivos técnicos

- Configurar entorno Django y conexión PostgreSQL.
- Implementar autenticación y sesiones (RF_1).
- Crear los primeros modelos (`User`, `Role`, `Course`) y sus migraciones.
- Definir esquema de permisos y restricciones por rol (RF_2, RF_3).
- Validar flujo de login y acceso a rutas restringidas mediante decoradores de rol.

Distribución de trabajo

Daniel Cely (CU_1, CU_2, CU_3)
Encargado del **núcleo de autenticación y evaluación inicial**. Implementará:

- RF_1: inicio de sesión con correo y autenticación segura.
- Configuración del *middleware* de sesiones y tokens.
- Flujo de acceso al sistema mediante Django Forms y HTMX.
  Su foco es establecer la infraestructura que conecta presentación con base de datos, preparando el terreno para los módulos posteriores de progreso y exámenes.

David Herrera (CU_1, CU_2, CU_3)
Responsable de la **gestión de usuarios y estructura de roles**.

- RF_2 y RF_3: CRUD de usuarios y gestión de permisos.
- Definición del modelo `UserRole` y manejo de estados activo/inactivo.
- Integración con el ORM y validación de consistencia.
  Esta implementación establece el control de acceso granular y los mecanismos de

auditoría inicial.

Daniel Gracia (CU_1, CU_2, CU_3)
Diseña la **gestión de roles y perfiles funcionales**, vinculando usuarios con rutas formativas iniciales.

- RF_3: configuración de roles vinculados a permisos en la UI.
- Estructuración del modelo `RoleAssignment`.
- Plantilla de interfaz base para acceso por tipo de usuario.
  Su trabajo asegura la coherencia del flujo de acceso y delimita el alcance de cada vista según rol.

Tomas Bermúdez (CU_1, CU_2, CU_3)
Construye el **módulo de cursos base** que luego se conectará con las rutas.

- RF_4: gestión de cursos (crear, modificar, publicar, retirar).
- Validaciones de publicación (solo analistas autorizados).
- Interfaz HTMX inicial para CRUD de cursos.
  Este módulo inaugura la interacción entre el ORM y la capa visual, sirviendo de modelo para las demás aplicaciones.

Semana 2: Expansión funcional — rutas, contenidos y evaluaciones

La segunda semana se centra en extender las funcionalidades hacia la **experiencia formativa y de evaluación**.
Aquí se consolida la relación entre cursos, contenidos y exámenes, y se implementan las rutas de aprendizaje que determinan la progresión secuencial del colaborador.
La meta es que el sistema ya sea navegable, con usuarios reales interactuando en un entorno controlado.

Objetivos técnicos

- Implementar CRUD completo para rutas y contenidos (RF_5, RF_6, RF_7).
- Añadir módulo de creación y carga de exámenes (RF_8, RF_9).

- Desarrollar interfaz de acceso del colaborador a sus cursos activos.
- Consolidar los servicios internos para búsquedas y notificaciones automáticas.

Distribución de trabajo

Tomas Bermúdez

- RF_6: gestión de rutas de aprendizaje. Implementará un *facade* que maneje el agregado `LearningPath` y sus cursos asociados.
- RF_11: supervisión del progreso, desarrollo del tablero de control para supervisores.
- Integración con señales para reflejar actualizaciones en tiempo real.

David Herrera

- RF_7: almacenamiento de materiales audiovisuales y documentos adjuntos.
- RF_9: carga masiva de preguntas desde archivo plano (TXT o JSON).
- Diseñará un *adapter* de almacenamiento (local → S3) y un *strategy* para validación de formatos.
  Su módulo amplía la persistencia y la interacción con archivos externos.

Daniel Cely

- RF_8: creación de exámenes y tareas, enlazados a cursos.
- Diseñará el *builder* de exámenes y la interfaz de creación.
- Implementará las notificaciones de tipo “nuevo examen creado” mediante el patrón *observador*.

Daniel Gracia

- RF_5: acceso del colaborador a sus rutas y cursos.
- RF_12: orden secuencial de contenidos (control de flujo entre módulos).
- Configurará el *iterator* y la *state machine* del progreso, de modo que el usuario avance por módulos según su orden asignado.

Semana 3: Integración, progreso y despliegue

La tercera semana integra todos los módulos en un ecosistema funcional.
El objetivo no es solo que los componentes “funcionen”, sino que **interactúen coherentemente**, con los eventos del sistema generando actualizaciones en cascada.
Aquí se valida la integridad de datos, el rendimiento y la cohesión visual.

Objetivos técnicos

- Consolidar los flujos de comunicación entre módulos (usuarios, cursos, evaluaciones, progreso).
- Implementar notificaciones automáticas basadas en eventos.
- Integrar la bitácora de auditoría y el registro de acciones críticas.
- Realizar pruebas de usabilidad y rendimiento.
- Documentar el proceso de despliegue (migraciones, dependencias, entorno).

Actividades por integrante

Daniel Cely
Consolida el flujo de registro y visualización de progreso (RF_10), conectando la información de exámenes completados con el tablero del supervisor.
Configura la capa de observación para emitir eventos `ProgressUpdated` y asegura su visualización en la interfaz del colaborador.

David Herrera
Integra los módulos de almacenamiento y carga masiva con el tablero administrativo, validando consistencia en las referencias de archivos.
Prepara el módulo para futuro escalamiento hacia un sistema de archivos distribuido.

Tomas Bermúdez
Lidera las pruebas de integración entre cursos, rutas y progreso supervisado.
Coordina el enlace entre módulos de dominio y el *dashboard* de supervisión, validando la correspondencia entre estados del curso y avance real.

Daniel Gracia
Asegura la correcta progresión de rutas formativas y secuencias (RF_12).
Implementa los *commands* finales para asegurar que la publicación o retiro de un curso actualice automáticamente las rutas y el progreso.

Herramientas de coordinación y control

El desarrollo se apoya en una infraestructura de colaboración moderna:

- **GitHub** como repositorio central, con ramas por módulo (`feature/accounts`, `feature/courses`, etc.).
- **Google Drive / Obsidian** para documentación estructurada (diagramas, requerimientos, reportes de avance).
- **Discord y WhatsApp** para comunicación operativa.
- **Trello o GitHub Projects** para seguimiento de tareas y asignación semanal.

Cada semana culmina con un *merge review* conjunto, donde se evalúan la calidad del código, la cobertura de requisitos y la integración visual.

Resultado esperado al cierre de la tercera semana

Al finalizar la planificación:

- Todos los usuarios podrán autenticarse y navegar sus rutas formativas.
- Los analistas tendrán control total sobre cursos, roles y evaluaciones.
- Los supervisores visualizarán el progreso consolidado de sus equipos.
- El sistema responderá de manera reactiva ante eventos (creación de curso, finalización de módulo, publicación de examen).

SAFE pasará de ser una estructura técnica a convertirse en un entorno de aprendizaje dinámico, donde cada acción desencadena respuestas automáticas dentro de un marco arquitectónico coherente.

---

3. Arquitectura técnica de SAFE (versión ampliada)

SAFE se implementará con **Django + PostgreSQL + HTMX** en una arquitectura por capas que separa con rigor presentación, dominio y persistencia. El corazón del diseño es

orientado a dominios de aprendizaje (usuarios/roles, cursos/contenidos, rutas, evaluaciones y progreso), y cada dominio se materializa como una *app* de Django con contratos claros y patrones de diseño específicos. Cuando corresponda, referencio los requisitos funcionales (RF) y casos de uso ya aprobados.

3.1 Capas y responsabilidades

Presentación (Django Templates + HTMX). Renderizado del lado servidor con fragmentos HTML reactivos. HTMX habilita interacciones asíncronas (hx-get/post/swap) para formularios, tableros y listas sin un SPA completo.
Negocio (Servicios de Dominio). Lógica de reglas, orquestación entre entidades y políticas de seguridad. Aislamos aquí la semántica de "quién puede hacer qué y en qué orden".
Persistencia (ORM de Django). Modelos, *Managers* y *QuerySets* especializados; transacciones atómicas y *migrations* versionadas.
Datos (PostgreSQL). Esquema normalizado (≈3FN), índices por claves externas y columnas de filtrado habitual (estado, fechas, rol). Disparadores se sustituyen preferentemente por señales en aplicación para mantener la lógica visible.

3.2 Módulos (apps) y contratos de integración

- **Identidad & Accesos.** `accounts/`
  Login y control de sesión (RF_1) con autorización por rol (RF_3) y estados de usuario (RF_2). Contrato: `AuthService` expone `authenticate()`, `authorize(role, action)` y emite eventos *UserLoggedIn*/*RoleChanged*.
  CU_1_dcelyi
  CU_1_dagraciap
  CU_1_daherreran

- **Cursos & Contenidos.** `courses/`
  CRUD de cursos y módulos, publicación/retirar (RF_4), almacenamiento de contenidos (RF_7) y orden secuencial dentro de cada módulo (RF_12). Contrato: `CourseService` gestiona ciclo de vida y secuencias.
  CU_1_tbermudezg
  CU_2_daherreran
  CU_3_dagraciap

- **Rutas de aprendizaje.** `paths/`
  Creación/edición de rutas, cálculo de duración y estados (RF_6). Contrato: `LearningPathService` agrega cursos y mantiene consistencia de orden.
  CU_2_tbermudezg

- **Evaluaciones.** `exams/`
  Creación de exámenes/tareas (RF_8) y carga masiva por TXT con validación de formato (RF_9). Contrato: `ExamService` + `QuestionImporter`.
  CU_2_dcelyi
  CU_3_daherreran

- **Experiencia del Colaborador.** `learner/`
  Acceso a rutas/cursos/contendido (RF_5), registro y recuperación del progreso (RF_10). Contrato: `ProgressService` y *facades* de catálogo.
  CU_2_dagraciap
  CU_3_dcelyi

- **Supervisión & Reportes.** `supervision/`
  Tablero para seguimiento de progreso (RF_11), filtros y acciones de reinscripción/alertas. Contrato: `DashboardReadModel` + `SupervisorActions`.
  CU_3_tbermudezg

Todos los contratos se comunican mediante **servicios** (clases de aplicación) y **eventos de dominio** publicados a través de señales de Django para desacoplar efectos colaterales (notificaciones, bitácora).

3.3 Patrones de diseño por componente (y por qué)

Presentación

- **MVT (Modelo–Vista–Template) con *Template Method* para vistas HTMX.**
  Estandarizamos vistas basadas en clases (`HXPartialView`) que implementan un *hook* `get_partial_context()`; así forzamos consistencia en fragmentos recargables (listas de cursos, formularios de usuario). Beneficio: menos duplicación en endpoints síncronos/HTMX.

- **Presenter/DTO para fragmentos.**
  Los objetos de dominio se proyectan a *view models* (DTO) con las mínimas columnas necesarias antes de renderizar templates parciales. Mejora rendimiento y evita fugas de campos sensibles.

- **Decorator para control de acceso fino.**
  Decoradores `@require_role('ANALYST')`, `@require_state('ACTIVE')` a nivel de vista, combinables con *permissions mixins* de Django. Esto mantiene la semántica de RF_2 y RF_3 visible en los puntos de entrada.
  CU_1_daherreran
  CU_1_dagraciap

Negocio (Servicios y Reglas)

- **Service Layer.**
  Cada *app* expone un servicio con operaciones atómicas (`assign_role`, `publish_course`, `add_content`, `enroll`, `record_progress`). Nos permite probar reglas sin tocar la web.

- **Domain Events + Observer (señales).**

  - *CoursePublished* → notifica a colaboradores suscritos y recalcula duración de rutas (RF_4, RF_6).

  - *ContentCompleted* → avanza *state machine* del progreso y emite *ProgressUpdated* (RF_10).

  - *ExamCreated*/*QuestionsImported* → envía alerta a responsables (RF_8, RF_9).
    Implementación: señales `post_save`/custom + *handlers* idempotentes.
    Beneficio: desacopla "qué ocurrió" de "qué hacer después".
    CU_1_tbermudezg
    CU_2_tbermudezg
    CU_3_dcelyi
    CU_3_daherreran

- **State (Máquina de estados) para progreso y publicaciones.**

  - Progreso: `INSCRITO → EN_CURSO → FINALIZADO` con transición a `TIEMPO_AGOTADO` por regla de fecha; cada transición valida precondiciones (RF_11).

  - Curso: `BORRADOR ↔ PUBLICADO ↔ RETIRADO` (RF_4).
    Esto elimina *ifs* dispersos y centraliza la lógica de transición.
    CU_3_tbermudezg
    CU_1_tbermudezg

- **Strategy para parsing/validación de archivos.**
  `QuestionImporter` elige la estrategia según *content-type*:

  - `PlainTextMCQStrategy` (formato con líneas, opciones A)/B)/C) y `*` en la correcta; límite de 20 ítems por lote) (RF_9).

  - `JSONBankStrategy` (extensible a futuro).
    Ventaja: nuevas fuentes sin tocar `ExamService`.
    CU_3_daherreran

- **Composite + Iterator para contenidos y módulos.**
  Un `Module` compone *ContentItems* (texto, video, PDF, quiz). Al recorrer la secuencia se aplica *Iterator* que respeta el orden configurado (RF_12) y permite "próximo"/"anterior" sin consultas complejas.
  CU_3_dagraciap

- **Specification para filtros ricos.**
  Filtros reutilizables (`ByStatus`, `ByCourse`, `ByTeam`) combinables (`&`, `|`) para el tablero de supervisor (RF_11) y catálogos (RF_5). Implementación: *QuerySet* enlazado a objetos *Specification*.

- **Facade para Rutas de aprendizaje.**
  `LearningPathFacade` agrupa operaciones: agregar/quitar curso, reordenar, recalcular duración, validar duplicados (RF_6). Disminuye el acoplamiento entre UI y varios servicios.
  CU_2_tbermudezg

- **Command + Audit Trail para operaciones sensibles.**
  Acciones como "asignar rol", "publicar curso", "cargar preguntas" se modelan como *Command* con `execute()` dentro de una transacción y registro de auditoría (quién, cuándo, antes/después). Cubre trazabilidad exigida en gestión de roles (RF_3) y publicaciones (RF_4/12).
  CU_1_dagraciap

- **Builder para exámenes.**
  `ExamBuilder` acumula metadata (nivel, duración, puntaje) y agrega *Question*s provenientes del *importer*. El *builder* valida consistencia (todas las preguntas con una sola correcta) antes de `build()` (RF_8/9).
  CU_2_dcelyi
  CU_3_daherreran

- **Policy/Guard Clauses para reglas de negocio.**
  Políticas explícitas: "no permitir retirar un curso asociado a rutas activas", "no permitir saltar contenido obligatorio" (RF_4, RF_12). Reducen ambigüedad y concentran validaciones.

Persistencia (ORM)

- **Repository (vía Managers) y Unit of Work (transacciones).**
  Managers especializados (`Course.objects.published()`, `Progress.objects.for_team(team_id)`) actúan como *repositories*. Las operaciones de servicios se encierran en `transaction.atomic()` como *Unit of Work*, garantizando consistencia entre varias escrituras (p.ej., inscribir + crear registro de progreso + emitir evento).

- **Aggregate Roots.**
    - `Course` (raíz de `Module` y `Content`).
    - `LearningPath` (raíz de `CourseInPath`).
    - `User` (raíz de `RoleAssignment`).
      Las escrituras pasan por la raíz para asegurar invariantes (no duplicar curso en ruta, mantener orden sin huecos).

- **Adapter para almacenamiento de archivos.**
  Interfaz `ContentStorage` con adaptadores: *LocalFileSystem* (desarrollo) y *S3-like* (a futuro). *Adapters* evitan acoplar el dominio a un proveedor.

- **Lazy Loading / Proxy para binarios.**
  Los campos `file` de *content/material* usan *FileField* con acceso diferido; la UI solo recupera metadatos salvo que el usuario consuma el recurso (RF_7).
  CU_2_daherreran

Transversales

- **Observer de Notificaciones.**
  Canaliza eventos del dominio a un `NotificationService` (correo/in-app). Ejemplos: *ExamCreated*, *DeadlineApproaching*, *ProgressStalled*. El patrón observador es el idóneo: los productores de eventos desconocen a los suscriptores, evitando acoplamiento y permitiendo añadir canales nuevos sin tocar el núcleo.

- **Cache-Aside para catálogos y conteos.**
  Listas frecuentes (catálogo de cursos publicados) y conteos del tablero se guardan en caché invalidada por eventos *CoursePublished/Retired* y *ProgressUpdated*.

- **Mapper/DTO y CQRS ligera.**
  Para vistas de lectura pesada (tablero del supervisor), usamos *read models* con consultas optimizadas y DTO planos; las escrituras siguen pasando por servicios del dominio. Evita N+1 y mantiene comandos limpios (RF_11).

- **Logger + Correlación.**
  ID de correlación por petición, *structured logging* (JSON) para enlazar comandos, consultas y señales; soporte de auditoría (RF_3/12).

- **Feature Flags.**
  Activar gradualmente importadores alternativos, nuevas reglas de secuencialidad o vistas HTMX sin *downtime*.

3.4 Flujo canónico extremo a extremo (ejemplo narrado)

1. **El Analista publica un curso (RF_4).** La vista protegida por `@require_role('ANALYST')` recibe la acción. `CourseService.publish(course_id)` valida estado (Policy), actualiza la entidad bajo `transaction.atomic()` y emite *CoursePublished*. El *observer* invalida cachés del catálogo y notifica a colaboradores suscritos.
   CU_1_tbermudezg

2. **La ruta se recalcula (RF_6).** *CoursePublished* es consumido por `LearningPathFacade`, que actualiza duraciones donde aparezca ese curso y garantiza que no haya duplicados u órdenes inconsistentes.
   CU_2_tbermudezg

3. **El Colaborador ve su panel (RF_5) y completa un contenido obligatorio (RF_12).** `ProgressService.complete(content_id, user_id)` consulta la *state machine* del módulo; si la política "no saltar obligatorios" se cumple, avanza a siguiente por *Iterator*, persiste y emite *ContentCompleted/ProgressUpdated*.
   CU_2_dagraciap
   CU_3_dagraciap

4. **El Supervisor observa el avance del equipo (RF_11).** La vista consulta el *read model* con *Specification* (`ByTeam & ByStatus(EN_CURSO)`), que está cacheado; los eventos de progreso lo invalidan selectivamente. Puede reenrolar con un *Command* auditado.
   CU_3_tbermudezg

5. **El Analista sube preguntas masivas (RF_9) y crea examen (RF_8).** `ExamService.create(...)` invoca `QuestionImporter.import(file)` que selecciona `PlainTextMCQStrategy`. `ExamBuilder` valida, genera el examen y publica *ExamCreated*. El *observer* envía notificación y el tablero refleja la evaluación pendiente.
   CU_3_daherreran
   CU_2_dcelyi

3.5 Decisiones técnicas clave y justificación

- **HTMX + Templates sobre SPA completo.** Minimiza complejidad de *build*, acelera *time-to-feature* y mantiene SEO y accesibilidad; el *Template Method* en vistas evita proliferación de controladores ad-hoc.

- **Eventos de dominio (Observer) para notificaciones y sincronías internas.** Evitan “llamadas en cadena” y habilitan crecimiento orgánico de funcionalidades reactivas.

- **State Machines explícitas.** El progreso y los estados de publicación son núcleos de negocio; modelarlos como estados evita regresiones y ambigüedades.

- **Strategy/Adapter en entradas/salidas variables.** Importadores y almacenamiento de archivos cambian con el tiempo; encapsular la variabilidad reduce deuda técnica.

- **Facade/Service Layer.** La UI conversa con *facades* de alto nivel (cursos, rutas, progreso), no con repositorios crudos; esto permite pruebas y cambios sin romper vistas.

3.6 Seguridad, rendimiento y operabilidad

- **Seguridad.** Autorización por rol y por recurso; *guards* de estado; sanitización de archivos (whitelist de extensiones y *content sniffing*); límites de tamaño y número de preguntas por lote (20) como parte del *Command* de importación (RF_9).
  CU_3_daherreran

- **Rendimiento.** Índices compuestos en `(user_id, course_id)` para progreso; *select_related/prefetch_related* en catálogos; caché con invalidación por eventos.

- **Operabilidad.** *Health checks* simples, *structured logging* y auditoría por *Command*. *Feature flags* para activar secuencialidad estricta en cursos ya publicados con confirmación (RF_12).
  CU_3_dagraciap

3.7 Resultado esperado de la arquitectura

Con esta arquitectura, SAFE asegura:

1. **Evolución controlada** (añadir nuevos tipos de contenido, evaluaciones o reglas de secuencia sin reescribir el núcleo).

2. **Experiencia fluida** con HTMX y fragmentos coherentes.

3. **Trazabilidad y auditoría** en operaciones sensibles (roles, publicaciones, exámenes).

4. **Lecturas eficientes** para supervisión y catálogos gracias a *read models*, *specifications* y caché.

En síntesis: el sistema queda preparado para crecer en funcionalidades sin quedar rehén de su propia complejidad, manteniendo los RF ya definidos y su semántica de negocio bajo control arquitectónico sólido.

1.

---

4. Aterrizaje técnico de la idea

Desde la perspectiva técnica, SAFE se desarrollará como un **ecosistema modular de aplicaciones Django**.
Cada módulo (usuarios, cursos, rutas, exámenes, progreso) será una aplicación autónoma interconectada mediante un *core* central que maneja autenticación, permisos y sesiones.

La **interfaz HTMX** permitirá llamadas asíncronas entre módulos sin recargar toda la vista, generando una sensación de fluidez propia de un sistema SPA (Single Page Application), pero manteniendo la ligereza y simplicidad del renderizado del lado del servidor.

En el diseño del flujo de datos:

- Las **vistas** actúan como puntos de entrada que procesan formularios y peticiones.
- Los **controladores** (views.py) coordinan la lógica y delegan las operaciones a los servicios de negocio.
- Los **servicios** se comunican con el ORM de Django y con las plantillas, aplicando validaciones de permisos, secuencia y progreso.
- La **comunicación entre módulos** usa señales de Django (`post_save`, `post_delete`), base de la implementación del patrón **observador**, que permitirá notificaciones automáticas al finalizar tareas o cursos.

El flujo completo mantiene la integridad de datos mediante el uso de **transacciones atómicas** y registros de auditoría que almacenan las acciones de los analistas en la bitácora del sistema.

---

5. Estrategia de validación e integración

Al cierre de la tercera semana, SAFE será validado mediante un conjunto de pruebas funcionales y de integración:

- Se simularán usuarios de diferentes roles para validar restricciones y permisos.
- Se probará la creación, carga y eliminación de contenidos.
- Se ejecutarán escenarios completos de evaluación, desde la carga de preguntas por TXT hasta la asignación de notas y la actualización del progreso.

El despliegue final se realizará en un entorno local, pero con estructura preparada para escalar hacia un entorno de producción (Docker o Render).

---

6. Conclusión

El desarrollo de SAFE combina la rigurosidad del diseño orientado a objetos con la flexibilidad del desarrollo web moderno.
Cada integrante del equipo implementará una pieza esencial del sistema, garantizando que las interacciones entre roles —Analista TH, Colaborador y Supervisor— se desarrollen dentro de un entorno controlado, seguro y modular.

> “SAFE no es solo una plataforma, sino un marco vivo de aprendizaje automatizado. Cada módulo refleja un aspecto del ciclo de desarrollo humano dentro de la organización: acceso, formación, evaluación y mejora continua.”