PLAN MAESTRO DE DESARROLLO PLAYTEST

# PROGRESO GENERAL

FASE 1: Infraestructura y Persistencia [██████░░░░] 60% - EN CURSO

FASE 2: Estadísticas y Consolidación [░░░░░░░░░░] 0% - PENDIENTE

FASE 3: Soporte Técnico y Comunicación [░░░░░░░░░░] 0% - PENDIENTE

FASE 4: Funcionalidades de Creadores [░░░░░░░░░░] 0% - PENDIENTE

FASE 5: Funcionalidades de Profesores [░░░░░░░░░░] 0% - PENDIENTE

FASE 6: Sistema de Luminarias [░░░░░░░░░░] 0% - PENDIENTE

FASE 7: Sistema de Torneos [░░░░░░░░░░] 0% - PENDIENTE

Progreso total:\*\* 18% completado (actualizado 15:30 - BD verificada)

# FASE 1: INFRAESTRUCTURA Y PERSISTENCIA DE JUEGOS

Objetivo: Asegurar que todos los juegos guardan correctamente datos y que la BD tiene todas las tablas necesarias.

## ETAPA 1.1: Auditoría y Verificación de Base de Datos

Descripción: Verificar qué tablas existen actualmente en Aiven y cuáles faltan según los endpoints del backend.

### Funcionalidad a desarrollar

- ✅ Lista completa de tablas existentes vs necesarias

- ✅ Identificación de columnas faltantes en tablas existentes

- ✅ Verificación de índices y constraints

### Cómo se va a hacer

1. ✅ Conectar a pgAdmin4 y exportar lista de tablas actuales

2. ✅ Revisar código backend para identificar tablas referenciadas

3. ✅ Comparar y crear lista de diferencias

4. ⏳ \*\*PENDIENTE:\*\* Consultar con usuario sobre estructura de tablas faltantes

## ETAPA 1.2 - Proponer estructura de tablas faltantes para aprobación.

Verificación mediante queries reveló que existen:

- ✅ Todas las tablas de JUEGOS (game\_players, game\_scores, user\_game\_configurations)

- ✅ Todas las tablas de SOPORTE (7 tablas)

- ✅ Todas las tablas de LUMINARIAS (5 tablas)

- ✅ Sistema de badges e insignias

- ✅ Sistema de niveles (level\_definitions, user\_levels, block\_levels)

- ✅ Sistema de comunicaciones

- ✅ Feature flags

- ✅ FAQ articles

- ✅ System logs

SOLO FALTA:

- ❌ Sistema de Torneos (5 tablas)

## ETAPA 1.3 - Verificar que modalidades guardan datos correctamente

Descripción

Crear las tablas `game\_players` y `game\_scores` si no existen, necesarias para el funcionamiento correcto de todas las modalidades.

### Funcionalidad a desarrollar

Persistencia completa de:

- Jugadores participantes en cada partida

- Puntuaciones finales por jugador y partida

- Métricas detalladas (tiempo, aciertos, errores)

### Cómo se va a hacer:

Paso 1:\*\* Verificar en pgAdmin4 si existen:

Paso 2:\*\* Propuesta de estructura (PENDIENTE DE APROBACIÓN):

Tabla `game\_players`

Tabla `game\_scores`

Paso 3:\*\* Tras aprobación del usuario, ejecutar en pgAdmin4

Paso 4:\*\* Verificar creación con:

### Partes críticas

- 🔴 \*\*MUY CRÍTICO:\*\* Foreign keys deben apuntar a tablas existentes (`games`, `users`)

- ⚠️ \*\*CRÍTICO:\*\* Columnas JSONB deben tener DEFAULT para evitar NULLs

- ⚠️ \*\*IMPORTANTE:\*\* Índices en `game\_id` y `user\_id` para performance

### Preguntas para el usuario

1. \*\*¿La estructura propuesta de `game\_players` es correcta?\*\*

- ¿Algún campo adicional necesario? (ej: status, left\_at, etc.)

- ¿El constraint UNIQUE(game\_id, user\_id) es adecuado?

2. \*\*¿La estructura propuesta de `game\_scores` es correcta?\*\*

- ¿El campo `score\_data` JSONB es suficientemente flexible?

- ¿Necesitas campos adicionales específicos fuera del JSONB?

3. \*\*¿Existen ya estas tablas con otro nombre en tu BD?\*\*

- Verificar si hay tablas similares con nombres distintos

### Afecciones a otras funcionalidades

- ✅ \*\*POSITIVA:\*\* Historial de juegos funcionará mejor

- ✅ \*\*POSITIVA:\*\* Rankings podrán calcularse correctamente

- ⚠️ \*\*SI FALLA:\*\* Algunas llamadas a `/api/games/history` darán error SQL hasta que se creen

### Testing post-implementación

1. ⏳ Verificar tablas creadas en pgAdmin4

2. ⏳ Insertar registro de prueba manualmente:

3. ⏳ Verificar que endpoint `/api/games` no da errores

4. ⏳ \*\*NO\*\* testing de funcionalidad aún, solo que no rompe nada

## ETAPA 1.3: Verificación de Guardado en Modalidades

Descripción: Aunque el análisis mostró que todas las modalidades tienen código de guardado, verificar que \*\*realmente funciona end-to-end\*\* con la BD.

### Funcionalidad a desarrollar

Testing exhaustivo de persistencia en las 9 modalidades.

### Cómo se va a hacer

Orden de testing (por prioridad):

\*\*MODALIDADES DE ENTRENAMIENTO:\*\*

1. ⏳ \*\*Classic Mode\*\* (`game-classic.html`)

2. ⏳ \*\*Streak Mode\*\* (`game-streak.html`)

3. ⏳ \*\*By Levels Mode\*\* (`game-by-levels.html`)

\*\*MODALIDADES DE COMPETICIÓN INDIVIDUAL:\*\*

4. ⏳ \*\*Exam Mode\*\* (`game-exam.html`)

5. ⏳ \*\*Lives Mode\*\* (`game-lives.html`)

6. ⏳ \*\*Time Trial Mode\*\* (`game-time-trial.html`)

7. ⏳ \*\*Marathon Mode\*\* (`game-marathon.html`)

\*\*MODALIDADES DE COMPETICIÓN MULTIPLAYER:\*\*

8. ⏳ \*\*Duel Mode\*\* (`game-duel.html`) - Requiere 2 usuarios

9. ⏳ \*\*Trivial Mode\*\* (`game-trivial.html`) - Requiere 2+ usuarios

Protocolo de testing por modalidad

Para CADA modalidad, ejecutar el siguiente checklist:

**A. Testing en Render (producción/staging)**

\*\*Paso 1: Preparación\*\*

- [ ] Login con usuario de prueba

- [ ] Abrir Developer Tools (F12)

- [ ] Ir a pestaña Network

- [ ] Activar filtro "Fetch/XHR"

\*\*Paso 2: Ejecución\*\*

- [ ] Iniciar partida de la modalidad

- [ ] Jugar hasta completar (o hasta condición de finalización)

- [ ] \*\*CAPTURAR:\*\*

- Screenshot de pantalla final

- Request/Response en Network tab

- Mensajes en Console tab

- Logs del backend en Render (Dashboard → Logs)

\*\*Paso 3: Verificación en pgAdmin4\*\*

**B. Checklist de validación**

Para que una modalidad se considere \*\*✅ FUNCIONAL\*\*, debe cumplir:

- [ ] Partida aparece en tabla `games` con `status = 'completed'`

- [ ] Score aparece en `game\_scores` con datos correctos en `score\_data`

- [ ] Jugador(es) aparecen en `game\_players`

- [ ] `user\_profiles.answer\_history` se actualiza (array crece)

- [ ] `user\_profiles.stats.consolidation` se actualiza (si aplica)

- [ ] Historial muestra la partida (`GET /api/games/history`)

- [ ] Frontend muestra mensaje de éxito al finalizar

**C. Categorías de estado**

\*\*✅ FUNCIONAL:\*\* Todos los checks pasan

\*\*⚠️ PARCIAL:\*\* Algunos checks fallan pero la partida se crea

\*\*❌ ROTO:\*\* La partida no se guarda en absoluto

\*\*❓ NO TESTEADO:\*\* Aún no se ha probado

**D. Registro de resultados**

| Modalidad | Estado | Issues encontrados | Prioridad fix |

|----------------|----------|---------------------------|------------------|

| Classic | ❓ | - | - |

| Streak | ❓ | - | - |

| By Levels | ❓ | - | - |

| Exam | ❓ | - | - |

| Lives | ❓ | - | - |

| Time Trial | ❓ | - | - |

| Marathon | ❓ | - | - |

| Duel | ❓ | - | - |

| Trivial | ❓ | - | - |

### Partes críticas

- ⚠️ \*\*CRÍTICO:\*\* Modalidades multiplayer (Duel, Trivial) requieren 2 cuentas de prueba

- ⚠️ \*\*CRÍTICO:\*\* `apiDataService.saveXXXScore()` puede fallar si tablas no existen

- ⚠️ \*\*IMPORTANTE:\*\* Verificar que no se duplican registros al guardar

- ⚠️ \*\*IMPORTANTE:\*\* Testing debe hacerse en Render, no en local

### Si algo falla durante testing

Protocolo de manejo de errores:\*\*

1. \*\*Capturar información:\*\*

- Error exacto en consola del navegador

- Status code de la respuesta HTTP

- Error en logs del backend (Render)

- Query SQL que falló (si aplica)

2. \*\*Identificar categoría:\*\*

- A. Frontend no envía datos → Problema en `game-\*.html`

- B. Backend rechaza request → Problema en `routes/games.js`

- C. BD rechaza INSERT → Problema en estructura de tablas

- D. Datos se guardan pero incorrectos → Problema en lógica

3. \*\*Reportar al plan:\*\*

- Actualizar tabla de estado

- Añadir nota en sección "Issues encontrados"

- Marcar para corrección en ETAPA 1.4

4. \*\*NO modificar código aún\*\* - Solo documentar

### Entregables

1. ⏳ Tabla completa con estado de guardado por modalidad (✅/⚠️/❌)

2. ⏳ Lista detallada de bugs encontrados (si hay)

3. ⏳ Logs de errores capturados con screenshots

4. ⏳ Recomendaciones de corrección priorizadas

## ETAPA 1.4: Corrección de Bugs de Guardado

Descripción: Si en ETAPA 1.3 encontramos modalidades que NO guardan correctamente, corregirlas una por una.

\*\*NOTA:\*\* Esta etapa podría no ser necesaria si todas las modalidades pasan el testing.

### Funcionalidad a desarrollar

Dependiendo de qué modalidades fallen, implementar correcciones específicas.

### Cómo se va a hacer

\*\*Proceso por modalidad rota:\*\*

**Paso 1: Análisis del fallo**

Para cada modalidad que falló en ETAPA 1.3:

A. Identificar tipo de fallo:\*\*

- [ ] A. Falta llamada a API → Añadir código

- [ ] B. API devuelve error → Investigar backend

- [ ] C. BD rechaza INSERT → Revisar constraints/tipos de datos

- [ ] D. Datos incorrectos → Revisar formato de datos

B. Localizar código problemático:\*\*

**Paso 2: Proponer solución**

ANTES de modificar código, documentar:

1. Archivo a modificar

2. Líneas afectadas

3. Código actual (problemático)

4. Código propuesto (solución)

5. Justificación del cambio

6. Posibles efectos secundarios

**Paso 3: Implementar corrección**

Solo después de aprobación del usuario:\*\*

1. Modificar archivo correspondiente

2. Commit con mensaje descriptivo:

3. Deploy a Render (si aplica)

4. Esperar a que deploy termine

**Paso 4: Testing de corrección**

Testing específico de modalidad corregida:\*\*

- [ ] Jugar 3 partidas completas

- [ ] Verificar que las 3 se guardan en BD

- [ ] Verificar que datos son correctos

- [ ] Verificar que stats se actualizan

- [ ] Verificar que historial muestra las partidas

Testing de regresión (otras modalidades):\*\*

- [ ] Jugar 1 partida de 2-3 modalidades que YA funcionaban

- [ ] Verificar que siguen funcionando correctamente

- [ ] Verificar que no hay efectos secundarios

**Paso 5: Actualizar registro**

Actualizar tabla de estado en ETAPA 1.3:

| Modalidad | Estado | Issues encontrados | Prioridad fix | Fix aplicado |

|-----------|--------|-------------------|---------------|--------------|

| Classic | ✅ CORREGIDO | No guardaba score | Alta | 8 Oct 2025 |

### Partes críticas

- 🔴 \*\*MUY CRÍTICO:\*\* Testing de regresión después de cada corrección

- ⚠️ \*\*CRÍTICO:\*\* No modificar lógica de juego, solo añadir guardado

- ⚠️ \*\*IMPORTANTE:\*\* Mantener consistencia en formato de `scoreData`

- ⚠️ \*\*IMPORTANTE:\*\* Un commit por modalidad corregida

### Patrones comunes de corrección

Patrón 1: Añadir guardado básico

Patrón 2: Actualizar estadísticas

Patrón 3: Manejo de errores

### Testing post-implementación

Por cada modalidad corregida:\*\*

1. \*\*Testing funcional:\*\*

- [ ] 3 partidas completas

- [ ] Verificar guardado en BD

- [ ] Verificar stats actualizadas

- [ ] Verificar historial

2. \*\*Testing de regresión:\*\*

- [ ] 1 partida de Classic (si no fue corregido)

- [ ] 1 partida de Duel (más complejo)

- [ ] Verificar que siguen funcionando

3. \*\*Testing de edge cases:\*\*

- [ ] Partida abandonada a mitad (debe NO guardarse)

- [ ] Sin conexión a internet (manejo de error)

- [ ] Partida muy corta (1 pregunta)

### Entregables

1. ⏳ Código corregido en repositorio

2. ⏳ Commits descriptivos por modalidad

3. ⏳ Tabla actualizada con estado de correcciones

4. ⏳ Documento de testing de regresión

# FASE 2: ESTADÍSTICAS Y CONSOLIDACIÓN DE CONOCIMIENTO

Objetivo: Sistema robusto de cálculo y visualización de estadísticas de aprendizaje.

## ETAPA 2.1: Auditoría de Sistema de Estadísticas Actual

Descripción. Entender exactamente cómo se calculan y guardan las estadísticas de consolidación actualmente.

### Funcionalidad a desarrollar

Documentación completa del sistema actual de stats.

### Cómo se va a hacer

1. Revisar estructura de `user\_profiles.stats`:

2. Revisar función de actualización:

- Analizar `routes/users.js:134-280` (función POST /stats)

- Documentar algoritmo de cálculo de consolidación

- Identificar parámetros configurables (pesos, umbrales)

3. Verificar qué modalidades actualizan stats:

- Buscar llamadas a `apiDataService.updateUserStats()` en cada modalidad

- Verificar formato de datos enviados

- Identificar inconsistencias

4. Testing actual del sistema:

5. Jugar partidas y observar:

- Jugar 3 partidas de Classic Mode

- Verificar si `stats` cambia después de cada una

- Documentar qué cambia exactamente

### Partes críticas

- ⚠️ \*\*IMPORTANTE:\*\* Fórmula de consolidación debe ser consistente

- ⚠️ \*\*IMPORTANTE:\*\* Verificar que no se pierden datos al actualizar JSONB

- ⚠️ \*\*IMPORTANTE:\*\* Entender diferencia entre `answer\_history` y `stats`

### Preguntas para el usuario

1. \*\*¿Qué representa exactamente `consolidationLevel`?\*\*

- ¿Es un porcentaje (0-1)?

- ¿Cómo se calcula? (aciertos/intentos, algoritmo más complejo, etc.)

- ¿Qué valor se considera "consolidado"? (ej: >0.8)

2. \*\*¿Qué es más importante para el sistema?\*\*

- A) Consolidación por pregunta individual

- B) Consolidación por tema

- C) Consolidación por bloque completo

3. \*\*¿El sistema actual funciona como esperas?\*\*

- ¿Te gustaría cambiar algo en cómo se calcula?

### Entregables

1. ⏳ Documento explicando algoritmo de consolidación actual

2. ⏳ Lista de modalidades que SÍ/NO actualizan stats

3. ⏳ Propuesta de mejoras (si usuario lo requiere)

4. ⏳ Ejemplos visuales de cómo cambian los datos

## ETAPA 2.2: Implementación de Actualización de Stats en Todas las Modalidades

Descripción: Asegurar que TODAS las modalidades actualizan correctamente `user\_profiles.stats` tras cada partida.

### Funcionalidad a desarrollar

Actualización consistente de:

- `consolidation.byQuestion` - Por cada pregunta respondida

- `consolidation.byTopic` - Por cada tema practicado

- `consolidation.byBlock` - Por cada bloque jugado

### Cómo se va a hacer

\*\*Identificar modalidades que NO actualizan:\*\*

Basándonos en ETAPA 2.1, crear lista:

- [ ] Classic Mode

- [ ] Streak Mode

- [ ] By Levels Mode

- [ ] Exam Mode

- [ ] Lives Mode

- [ ] Time Trial Mode

- [ ] Marathon Mode

- [ ] Duel Mode

- [ ] Trivial Mode

Por cada modalidad que NO actualice:\*\*

Paso 1: Preparar datos de respuestas

Paso 2: Llamar a updateUserStats

Paso 3: Verificar en BD

### Partes críticas

- 🔴 \*\*MUY CRÍTICO:\*\* Formato de `answerHistory` debe ser exacto

- ⚠️ \*\*CRÍTICO:\*\* No duplicar llamadas a `updateUserStats` (una vez por partida)

- ⚠️ \*\*IMPORTANTE:\*\* Manejar partidas abandonadas (no actualizar stats si no termina)

- ⚠️ \*\*IMPORTANTE:\*\* Performance si partidas muy largas (muchas preguntas)

### Casos especiales por modalidad

\*\*Marathon Mode:\*\*

- Preguntas que no se llegaron a ver NO cuentan

\*\*Duel/Trivial Mode:\*\*

- Stats se actualizan para AMBOS jugadores

\*\*Exam Mode:\*\*

- Preguntas en blanco NO cuentan como intento

### Testing post-implementación

Por cada modalidad implementada:\*\*

1. \*\*Testing básico:\*\*

- [ ] Jugar partida con 10 preguntas conocidas

- [ ] Verificar que 10 preguntas aparecen en stats

- [ ] Verificar que consolidación se calcula

2. \*\*Testing de acumulación:\*\*

- [ ] Jugar 3 partidas con mismas preguntas

- [ ] Verificar que `attempts` aumenta

- [ ] Verificar que `consolidationLevel` cambia

3. \*\*Testing de edge cases:\*\*

- [ ] Partida abandonada → NO actualizar

- [ ] Partida con 1 pregunta → Actualizar correctamente

- [ ] Partida con preguntas nuevas → Crear entradas nuevas

### Entregables

1. ⏳ Código actualizado en todas las modalidades

2. ⏳ Testing documentado por modalidad

3. ⏳ Tabla de estado de implementación

4. ⏳ Métricas de performance (si aplica)

## ETAPA 2.3: Endpoint de Estadísticas Detalladas

Descripción: Crear/mejorar endpoint que devuelva estadísticas procesadas y listas para mostrar en frontend.

### Funcionalidad a desarrollar

Endpoint `GET /api/users/stats/detailed` que devuelva:

- Preguntas más débiles (menor consolidación)

- Temas dominados vs pendientes

- Progreso por bloque

- Recomendaciones de estudio personalizadas

- Estadísticas globales del usuario

### Cómo se va a hacer:

**Paso 1: Crear endpoint en `routes/users.js`**

1. Obtener stats del usuario

2. Procesar preguntas débiles (consolidación < 0.5)

3. Procesar progreso por tema

4. Procesar progreso por bloque

5. Calcular consolidación general

6. Generar recomendaciones

Estadísticas adicionales

Recomendación 1: Preguntas débiles

Recomendación 2: Temas pendientes

Recomendación 3: Bloques a repasar

Recomendación 4: Consolidación general baja

Recomendación 5: Consolidación alta - mantener

**Paso 2: Testing del endpoint**

# Testing en local

# Testing en Render

### Partes críticas

- ⚠️ \*\*CRÍTICO:\*\* Validar que `stats.consolidation` existe antes de procesar

- ⚠️ \*\*IMPORTANTE:\*\* Performance con muchas preguntas (limitar a top N)

- ⚠️ \*\*IMPORTANTE:\*\* Manejar caso de usuario nuevo (sin datos)

### Preguntas para el usuario

1. \*\*¿Qué umbrales prefieres para los estados?\*\*

- Actual propuesto:

- Mastered: ≥ 0.8

- Good: 0.6-0.79

- Learning: 0.4-0.59

- Weak: < 0.4

2. \*\*¿Cuántas recomendaciones máximo quieres mostrar?\*\*

- Propuesta: Top 5 más importantes

3. \*\*¿Qué otras estadísticas te gustaría ver?\*\*

- Racha actual/máxima

- Tiempo total estudiado

- Preguntas favoritas

- Etc.

### Testing post-implementación

1. ⏳ Verificar respuesta tiene estructura correcta

2. ⏳ Verificar cálculos con casos conocidos

3. ⏳ Testing con usuario sin datos (nuevo)

4. ⏳ Testing con usuario con muchos datos (performance)

5. ⏳ Verificar recomendaciones son coherentes

### Entregables

1. ⏳ Endpoint implementado y funcional

2. ⏳ Documentación de API

3. ⏳ Tests unitarios (opcional pero recomendado)

4. ⏳ Ejemplos de respuestas

## ETAPA 2.4: Frontend de Visualización de Estadísticas

Descripción: Crear/mejorar panel de estadísticas para que usuario vea su progreso de consolidación de forma clara e intuitiva.

### Funcionalidad a desarrollar

Panel con:

- 📊 Gráfico de consolidación por tema (barras)

- 📋 Lista de preguntas débiles (tabla interactiva)

- 🎯 Progreso por bloque (circular/radial)

- 💡 Recomendaciones personalizadas (cards)

- 📈 Estadísticas globales (resumen)

### Cómo se va a hacer

Paso 1: Crear/mejorar archivo HTML

\*\*Opciones:\*\*

- A) Crear nuevo `statistics-dashboard.html`

- B) Mejorar existente si hay alguno

Paso 2: Estructura base del panel

Paso 3: Consumir endpoint de estadísticas

Paso 4: Renderizar sección de resumen

Paso 5: Renderizar recomendaciones

Paso 6: Renderizar gráfico de temas (Chart.js)

Paso 7: Renderizar tabla de preguntas débiles

Paso 8: Renderizar progreso por bloque

### Partes críticas

- ⚠️ \*\*CRÍTICO:\*\* Manejar caso de usuario nuevo sin estadísticas

- ⚠️ \*\*IMPORTANTE:\*\* Performance con muchos datos (limitar visualización)

- ⚠️ \*\*IMPORTANTE:\*\* Chart.js debe cargarse correctamente

- ⚠️ \*\*IMPORTANTE:\*\* Responsive design para móviles

### Nuevos endpoints necesarios

1. Endpoint para obtener textos de preguntas por IDs:\*\*

2. Endpoint para obtener nombres de bloques por IDs:\*\*

### Preguntas para el usuario

1. \*\*¿Qué diseño prefieres para el dashboard?\*\*

- A) Todo en una página (scroll vertical)

- B) Pestañas (resumen, temas, preguntas, bloques)

- C) Sidebar con navegación

2. \*\*¿Qué gráficos son más importantes?\*\*

- Barras de temas (propuesto)

- Líneas de evolución temporal

- Radar/spider de múltiples dimensiones

- Circular/donut de distribución

3. \*\*¿Necesitas exportar estadísticas?\*\*

- PDF

- CSV

- Imágenes

### Testing post-implementación

1. ⏳ Verificar que todos los componentes se renderizan

2. ⏳ Probar con usuario nuevo (sin datos)

3. ⏳ Probar con usuario con muchos datos

4. ⏳ Verificar responsive (móvil, tablet, desktop)

5. ⏳ Verificar que botones de "Practicar" funcionan

6. ⏳ Verificar que gráficos son interactivos

### Entregables

1. ⏳ Archivo HTML completo y funcional

2. ⏳ Endpoints `/batch` implementados

3. ⏳ Screenshots del dashboard funcionando

4. ⏳ Documentación de uso para usuario final

### Notas y comentarios

Esta es la parte más visible para el usuario. Requiere especial atención al diseño y UX. Considerar feedback del usuario sobre disposición de elementos.

Posible mejora futura: Actualización en tiempo real con WebSockets.

# FASE 3: SOPORTE TÉCNICO Y COMUNICACIÓN

Objetivo:\*\* Sistema funcional de tickets de soporte y comunicación interna.

## ETAPA 3.1: Creación de Tablas de Soporte

Descripción: Ejecutar script que crea todas las tablas del sistema de soporte técnico.

### Funcionalidad a desarrollar

Infraestructura de BD para:

- Tickets de soporte con estados y prioridades

- Categorías de problemas predefinidas

- Comentarios/conversaciones en tickets

- Agrupación inteligente de tickets similares

- Base de conocimiento (KB) con artículos

- Plantillas de respuestas rápidas

- Sistema de escalación automática

- Métricas y dashboard para agentes

### Cómo se va a hacer

**Paso 1: Revisar script existente**

Archivo: `playtest-backend/update-support-schema.js`

\*\*Verificar qué crea:\*\*

- ⏳ Listar todas las tablas que creará

- ⏳ Revisar estructura de cada tabla

- ⏳ Identificar foreign keys y dependencias

**Paso 2: Consultar al usuario**

\*\*PREGUNTAS IMPORTANTES - RESPONDER ANTES DE EJECUTAR:\*\*

1. \*\*¿Quiénes pueden crear tickets de soporte?\*\*

- A) Todos los usuarios

- B) Solo usuarios con rol específico

- C) Solo usuarios verificados/pagados

2. \*\*¿Quiénes serán agentes de soporte?\*\*

- A) Administradores (principal y secundario)

- B) Rol específico "soporte" (crear nuevo rol)

- C) Creadores de contenido también

3. \*\*¿Qué categorías de tickets necesitas?\*\*

Propuesta inicial:

- Problema técnico (bug)

- Pregunta sobre funcionalidad

- Sugerencia/mejora

- Problema con contenido

- Problema de cuenta

- Otro

4. \*\*¿Qué prioridades de tickets?\*\*

Propuesta:

- Crítica (sistema caído, pérdida de datos)

- Alta (funcionalidad no disponible)

- Media (inconveniente menor)

- Baja (pregunta, sugerencia)

5. \*\*¿Escalación automática?\*\*

- ¿Después de cuántas horas sin respuesta?

- ¿A quién escalar? (admin principal, etc.)

**Paso 3: Modificar script si necesario**

Basándose en respuestas del usuario, ajustar:

- Categorías predefinidas

- Roles con permisos

- Reglas de escalación

- Plantillas iniciales

**Paso 4: Ejecutar script**

**Paso 5: Verificar creación**

**Paso 6: Insertar datos iniciales**

### Estructura de tablas (propuesta)

\*\*1. support\_tickets\*\*

\*\*2. support\_categories\*\*

\*\*3. support\_comments\*\*

\*\*4. support\_ticket\_groups\*\*

\*\*5. support\_knowledge\_base\*\*

\*\*6. support\_templates\*\*

\*\*7. support\_escalations\*\*

\*\*8. support\_dashboard\_metrics\*\*

### Partes críticas

- 🔴 \*\*MUY CRÍTICO:\*\* Foreign keys a `users` deben existir

- ⚠️ \*\*CRÍTICO:\*\* Rol de soporte debe existir en tabla `roles` (o crear agentes con rol admin)

- ⚠️ \*\*IMPORTANTE:\*\* Triggers de escalación automática (si se implementan)

- ⚠️ \*\*IMPORTANTE:\*\* Índices en campos de búsqueda frecuente

### Testing post-implementación

1. ⏳ Verificar todas las tablas creadas en pgAdmin4

2. ⏳ Verificar foreign keys y constraints:

3. ⏳ Insertar ticket de prueba manualmente:

4. ⏳ Verificar que triggers funcionan (ticket\_number se genera)

5. ⏳ Verificar que métricas se actualizan

### Entregables

1. ⏳ Todas las tablas de soporte creadas en Aiven

2. ⏳ Categorías predefinidas insertadas

3. ⏳ Plantillas iniciales insertadas

4. ⏳ Documento de verificación con queries y resultados

5. ⏳ Rol "soporte" creado (si aplica)

### Notas y comentarios

Sistema de soporte es completamente independiente. Puede implementarse en paralelo

con otras fases si se desea.

Backend de soporte (routes/support.js) ya está completamente implementado con

896 líneas de código. Solo falta crear las tablas.

## ETAPA 3.2: Testing de Endpoints de Soporte

Descripción: Verificar que los endpoints de `routes/support.js` funcionan correctamente con las tablas creadas.

### Funcionalidad a desarrollar

Testing exhaustivo de:

- Creación de tickets

- Listado con filtros múltiples

- Sistema de comentarios

- Asignación de agentes

- Cambios de estado y prioridad

- Búsqueda y agrupación

- Base de conocimiento

- Plantillas

- Métricas dashboard

### Cómo se va a hacer

Testing con herramientas API

\*\*Opciones:\*\*

1. Postman (recomendado - permite guardar colecciones)

2. cURL (más rápido para tests simples)

3. Thunder Client (extensión VSCode)

4. Insomnia

### Preparación

\*\*1. Obtener token de autenticación:\*\*

\*\*2. Guardar token en variable:\*\*

#### Suite de tests

\*\*TEST 1: Crear ticket básico\*\*

\*\*TEST 2: Listar todos los tickets\*\*

\*\*TEST 3: Listar tickets con filtros\*\*

\*\*TEST 4: Obtener detalles de un ticket\*\*

\*\*TEST 5: Añadir comentario a ticket\*\*

\*\*TEST 6: Cambiar estado del ticket\*\*

\*\*TEST 7: Cambiar prioridad\*\*

\*\*TEST 8: Asignar ticket a agente\*\*

\*\*TEST 9: Buscar en base de conocimiento\*\*

\*\*TEST 10: Obtener plantillas\*\*

\*\*TEST 11: Dashboard de métricas\*\*

\*\*TEST 12: Resolver y cerrar ticket\*\*

### Checklist de validación

Para cada endpoint, verificar:

\*\*Checklist general:\*\*

- [ ] Status code correcto (200, 201, 400, 403, 500)

- [ ] Respuesta JSON válida

- [ ] Campos esperados presentes

- [ ] Tipos de datos correctos

- [ ] Sin errores en logs del servidor

\*\*Checklist de permisos:\*\*

- [ ] Usuario normal puede crear tickets

- [ ] Usuario normal puede ver sus propios tickets

- [ ] Usuario normal NO puede ver tickets de otros

- [ ] Agente/Admin puede ver todos los tickets

- [ ] Agente/Admin puede asignar y cambiar estados

- [ ] Solo Admin puede acceder a configuración

\*\*Checklist de validación de datos:\*\*

- [ ] Subject vacío → Error 400

- [ ] Category inválido → Error 400

- [ ] Priority inválido → Error 400

- [ ] Status inválido → Error 400

- [ ] ID no existente → Error 404

\*\*Checklist de BD:\*\*

- [ ] Datos se guardan correctamente en BD

- [ ] Foreign keys mantienen integridad

- [ ] Timestamps se actualizan

- [ ] Triggers funcionan (ticket\_number, métricas)

### Tabla de resultados de testing

| Endpoint | Método | Status | Issues | Notas |

|----------|--------|--------|--------|-------|

| `/tickets` (crear) | POST | ⏳ | - | - |

| `/tickets` (listar) | GET | ⏳ | - | - |

| `/tickets` (filtros) | GET | ⏳ | - | - |

| `/tickets/:id` | GET | ⏳ | - | - |

| `/tickets/:id/comments` | POST | ⏳ | - | - |

| `/tickets/:id/status` | PATCH | ⏳ | - | - |

| `/tickets/:id/priority` | PATCH | ⏳ | - | - |

| `/tickets/:id/assign` | PATCH | ⏳ | - | - |

| `/knowledge-base` | GET | ⏳ | - | - |

| `/templates` | GET | ⏳ | - | - |

| `/dashboard/metrics` | GET | ⏳ | - | - |

### Partes críticas

- ⚠️ \*\*CRÍTICO:\*\* Verificar permisos (usuarios normales vs agentes)

- ⚠️ \*\*IMPORTANTE:\*\* Validación de datos en endpoints

- ⚠️ \*\*IMPORTANTE:\*\* Testing con usuario sin permisos de agente

- ⚠️ \*\*IMPORTANTE:\*\* Verificar que métricas se actualizan en tiempo real

### Si algo falla

\*\*Protocolo:\*\*

1. \*\*Capturar error completo:\*\*

2. \*\*Identificar categoría:\*\*

3. \*\*Documentar y reportar:\*\*

### Testing post-implementación

1. ⏳ Todos los endpoints básicos funcionan

2. ⏳ Filtros y búsqueda funcionan

3. ⏳ Permisos correctos

4. ⏳ BD se actualiza correctamente

5. ⏳ Métricas dashboard actualizadas

### Entregables

1. ⏳ Tabla completa de resultados de testing

2. ⏳ Colección Postman (opcional pero recomendado)

3. ⏳ Lista de issues encontrados

4. ⏳ Screenshots de respuestas exitosas

5. ⏳ Queries de verificación en BD

### Notas y comentarios

Esta etapa es puramente de testing. No se modifica código.

Recomendación: Crear colección Postman con todos los tests para poder repetirlos fácilmente después de cambios futuros.

Considerar crear script de testing automatizado con Jest o similar.

## ETAPA 3.3: Integración Frontend de Soporte

Descripción: Conectar archivos HTML de soporte con el backend funcional. Crear flujos completos para usuarios y agentes.

### Funcionalidad a desarrollar

Frontend completo de:

- `support-form.html` - Formulario para crear ticket (usuarios)

- `support-tickets.html` - Lista y gestión de tickets (usuarios + agentes)

- `support-dashboard.html` - Dashboard para agentes

- `support-knowledge.html` - Base de conocimiento (público)

- Integración en header/menú principal

### Cómo se va a hacer

\*(Se continuará detallando cuando se llegue a esta etapa)\*

## FASES SIGUIENTES (4-7)

Las siguientes fases serán detalladas completamente cuando estemos cerca de implementarlas:

FASE 4: FUNCIONALIDADES DE CREADORES DE CONTENIDO

- Gestión avanzada de bloques y preguntas

- Panel de analytics para creadores

- Sistema de valoraciones de contenido

- Marketplace de bloques (conexión con Luminarias)

FASE 5: FUNCIONALIDADES DE PROFESORES

- Gestión de alumnos/grupos

- Asignación de bloques/tareas

- Seguimiento de progreso de alumnos

- Reportes y estadísticas de clase

FASE 6: SISTEMA DE LUMINARIAS

- Configuración de valores y reglas

- Tienda de items/servicios

- Marketplace entre usuarios

- Sistema de conversión a dinero real

- Panel administrativo de Luminarias

FASE 7: SISTEMA DE TORNEOS

- Creación y configuración de torneos

- Inscripción y gestión de participantes

- Sistema de brackets/eliminatorias

- Seguimiento en tiempo real

- Distribución de premios

## GESTIÓN DEL PLAN

### Cómo usar este documento

**Para consultar progreso:**

1. Ir a sección "PROGRESO GENERAL" al inicio

2. Ver barra de progreso de cada fase

3. Buscar "⬅️ ESTAMOS AQUÍ" para ver dónde estamos

**Para trabajar en una etapa:**

1. Localizar la etapa actual (buscar "EN CURSO")

2. Leer "Descripción" y "Cómo se va a hacer"

3. Ejecutar pasos en orden

4. Actualizar checkboxes [ ] a [x] al completar

5. Añadir comentarios en "Notas y comentarios"

6. Cambiar estado a "COMPLETADA" al terminar

**Para reportar problemas:**

1. En la etapa correspondiente, sección "Notas y comentarios"

2. Añadir entrada con formato:

```

[FECHA - HORA] Usuario/Claude:

PROBLEMA: Descripción del problema

CONTEXTO: Qué se estaba haciendo

ERROR: Mensaje de error exacto (si aplica)

SOLUCIÓN APLICADA: (si se resolvió)

PENDIENTE: (si queda algo por hacer)

```

**Para actualizar progreso:**

1. Cambiar estado de etapa de "EN CURSO" a "COMPLETADA"

2. Actualizar fecha de finalización

3. Actualizar barra de progreso de la fase ([████░░] añadir más █)

4. Actualizar "PROGRESO GENERAL" al inicio

5. Commit cambios con mensaje descriptivo

### Convenciones

Estados:

- ✅ \*\*COMPLETADA\*\* - Terminada y verificada

- 🟢 \*\*COMPLETADA CON OBSERVACIONES\*\* - Terminada pero con notas

- 🟡 \*\*EN CURSO\*\* - Trabajando actualmente

- ⏳ \*\*PENDIENTE\*\* - No iniciada aún

- 🔴 \*\*BLOQUEADA\*\* - No puede iniciarse por dependencias

- ⚠️ \*\*CON PROBLEMAS\*\* - Iniciada pero con issues

Prioridades:

- 🔴 \*\*MUY CRÍTICO\*\* - Puede romper todo el sistema

- ⚠️ \*\*CRÍTICO\*\* - Puede causar pérdida de datos o bugs graves

- ⚠️ \*\*IMPORTANTE\*\* - Debe hacerse bien pero no rompe sistema

- ℹ️ \*\*INFO\*\* - Información relevante pero no crítica

Símbolos de progreso:

- █ - Completado

- ░ - Pendiente

- ⬅️ - Estamos aquí