ChatGPT

Plan de Proyecto: Plataforma de Carpooling “BlaBlaCar Venezuela” (Spec-Driven Development)

Visión General del Proyecto

Este proyecto consiste en desarrollar una plataforma de **carpooling** inspirada en BlaBlaCar, adaptada al mercado venezolano. Será una aplicación web donde **conductores** con asientos disponibles puedan publicar viajes interurbanos, y **pasajeros** puedan buscar esos viajes y reservar un asiento compartiendo los gastos de transporte. BlaBlaCar ha demostrado la viabilidad de este modelo: conecta conductores con pasajeros para viajes de media/larga distancia, permite compartir costos, y se apoya en perfiles verificados, calificaciones y reseñas para crear confianza en la comunidad [1]. Nuestra adaptación local deberá considerar las particularidades del mercado venezolano (por ejemplo, métodos de pago disponibles, conectividad móvil limitada, seguridad) sin perder las funcionalidades clave: registro de usuarios, publicación/búsqueda de viajes, reservas, sistema de reputación y comunicación entre usuarios [2].

Objetivos: Crear un **Producto Mínimo Viable (MVP)** que cubra las funciones esenciales de la plataforma BlaBlaCar adaptadas a Venezuela, asegurando una experiencia sencilla y segura para los usuarios. Como único fundador y desarrollador, se adoptarán metodologías ágiles y enfoque *Spec-Driven Development* para gestionar eficientemente el proyecto. Esto significa que antes de codificar se definirá con detalle **qué debe hacer el sistema** (requisitos funcionales y técnicos), plasmándolo en especificaciones claras, de forma que dichas especificaciones se conviertan en el artefacto central que guía todo el ciclo de desarrollo [3]. En resumen, primero se documentará **el “qué y cómo” del sistema** (incluyendo criterios de aceptación y diseño técnico), y luego se implementará el código basado en esas especificaciones, apoyándose en herramientas modernas (como agentes GPT/Codex) para acelerar la codificación. Este enfoque garantiza mayor claridad en requisitos, mejor alineación (aunque seamos solo un desarrollador, las especificaciones sirven para planificar y verificar) y un código más consistente y mantenible [4].

Alcance y Funcionalidades Principales

A continuación se definen las características principales del sistema, organizadas por tipo de usuario y módulo. Estas funciones se derivan del modelo de BlaBlaCar y se ajustan al contexto local. Servirán de base para redactar las especificaciones detalladas de cada componente en la etapa de *Spec-Driven Development*.

- **Registro y Autenticación de Usuarios:** Se utilizará Supabase Auth para manejar la creación de cuentas, inicio de sesión y gestión de sesiones. Los usuarios podrán registrarse con email y contraseña (y potencialmente OAuth en el futuro). Dado que Supabase ofrece autenticación basada en **JSON Web Tokens (JWT)**, la app aprovechará esa capa: al iniciar sesión, el front-end obtendrá un JWT de Supabase. En cada solicitud subsecuente al backend, ese token se enviará en la cabecera de autorización, y el servidor FastAPI lo verificará para autenticar al usuario [5] [6]. Esto asegura que solo usuarios autenticados puedan acceder a las funciones sensibles (publicar viajes, reservar, etc.). Tras registrarse, cada usuario tendrá un **perfil** asociado en la base de datos donde podrá completar

información adicional (nombre, teléfono, foto, preferencias, etc.), ya que Supabase Auth por defecto provee solo autenticación básica.

- **Panel de Pasajero (Buscador de Viajes):** Un usuario de tipo pasajero puede buscar viajes disponibles. Las opciones de búsqueda incluirán origen, destino y fecha. El sistema mostrará una lista de viajes publicados que coincidan, con detalles como el conductor, horario, precio por asiento y calificaciones del conductor. Desde la perspectiva de UI (interfaz de usuario), habrá páginas de listado de resultados y una página de detalle de viaje. Desde la perspectiva funcional, esto implica en el backend un **endpoint** para consultar viajes filtrados. Los pasajeros podrán **reservar un asiento** en un viaje disponible: esto creará una **reserva** en la base de datos asociando el pasajero con el viaje y reduciendo la disponibilidad de asientos. Idealmente, la reserva podría requerir confirmación del conductor (según modelo de negocio), pero en el MVP podemos asumir que la reserva se confirma automáticamente. Finalmente, tras completar un viaje, los pasajeros podrán **calificar y reseñar** al conductor (y viceversa), alimentando un sistema de **reputación**. Estas calificaciones y reseñas estarán visibles en los perfiles, ayudando a generar confianza en la comunidad [1].

- **Panel de Conductor (Publicación de Viajes):** Un usuario de tipo conductor puede publicar la oferta de un viaje. Para ello deberá proporcionar información como ciudad de origen, ciudad de destino, fecha y hora de salida, número de asientos disponibles, costo por pasajero (en la moneda local, p. ej. Bolívares o dólares), y opcionalmente detalles del vehículo o normas (ej: “no fumar”). Cada publicación de viaje se guarda en la base de datos. Los conductores pueden ver la lista de sus viajes publicados (pasados y futuros) y la lista de reservas recibidas para cada viaje. Deben poder **aprobar o rechazar** solicitudes de reserva si implementamos moderación, o simplemente ser notificados cuando alguien reserve (para el MVP podemos suponer confirmación automática). Tras el viaje, el conductor también puede calificar a los pasajeros.

- **Comunicación entre Usuarios:** Es importante que pasajeros y conductores puedan comunicarse para coordinar detalles (punto de encuentro, etc.). En BlaBlaCar típicamente existe una mensajería interna. En nuestro MVP podríamos implementar un **sistema simple de mensajería** dentro de cada viaje reservado: por ejemplo, un chat asociado a la reserva. Alternativamente, inicialmente podríamos exponer un método de contacto (teléfono/WhatsApp) del conductor una vez realizada la reserva, pero eso implica compartir datos sensibles. Idealmente, un módulo de chat en tiempo real (usando, por ejemplo, Supabase Realtime o WebSockets en FastAPI) sería lo deseable en versiones futuras. Para la primera versión, podemos simplificarlo a permitir que el pasajero envíe un mensaje de texto corto al conductor a través de la plataforma (almacenado en BD) y que el conductor reciba una notificación por email o dentro de su panel.

- **Sistema de Pago (Consideraciones locales):** BlaBlaCar normalmente implementa pago electrónico adelantado para asegurar la reserva [2]. En el contexto local, habrá que decidir si integrar pagos en la aplicación o manejar pagos offline (efectivo). Para un MVP, podríamos posponer la integración de pasarelas de pago complejas y optar por un modelo donde el pasajero paga en efectivo al conductor el día del viaje, pero se registra la intención de pago en la plataforma. Sin embargo, para escalabilidad, es recomendable soportar pagos en línea. Supabase no ofrece pagos, así que en el futuro podríamos integrar un proveedor local (ej: pago móvil, PayPal, Zelle). **En el documento de especificación se delineará esta decisión**; si se omiten pagos online en MVP, las especificaciones de reserva indicarán que la plataforma solo coordina la reserva pero no transacciona dinero. Aún así,

es importante registrar el estado de pago de una reserva (pagado/no pagado) por si luego se integra un mecanismo de confirmación de pago.

- **Panel de Administración:** Aunque siendo un solo desarrollador inicialmente no habrá un equipo grande operando la plataforma, es útil prever un **panel admin** (back-office) mínimo para moderación y soporte. Este panel permitiría ver listado de usuarios, viajes publicados, reservas, y posibilitar acciones administrativas: eliminar o editar publicaciones inapropiadas, bloquear usuarios que incumplan normas, etc. Dado el alcance limitado, podría no desarrollarse una interfaz web separada al inicio, sino usar directamente herramientas sobre la base de datos (o pequeñas vistas dentro de la misma app con permiso de administrador). De todos modos, en la especificación se anotará la necesidad de estas capacidades administrativas para garantizar la **calidad del servicio y seguridad** de la plataforma.

Requerimientos Funcionales Desglosados

A continuación, se enumeran en forma de **historias de usuario** algunos de los requerimientos funcionales principales, que luego servirá para crear archivos de especificaciones detallados por funcionalidad (por ejemplo, `1.registro.spec.md`, `2.publicar_viaje.spec.md`, etc., siguiendo el enfoque SDD de 1 o más pasos). Cada requerimiento deberá incluir criterios de aceptación claros.

- **Registro de Usuario:** *"Como visitante, quiero poder crear una cuenta de usuario proporcionando mi email y contraseña, para poder acceder a las funcionalidades de la plataforma. Debo confirmar mi dirección de correo (por ejemplo, via un enlace de verificación enviado) antes de poder usar completamente la aplicación."* **Criterios de Aceptación:** Validación de email único, contraseña segura; posibilidad de recuperar contraseña; integración con Supabase Auth; perfil inicial creado tras registro.

- **Inicio de Sesión y Cierre:** *"Como usuario registrado, quiero iniciar sesión con mi email/contraseña para acceder a mi cuenta, y quiero poder cerrar sesión de forma segura."* **Criterios:** Al iniciar, obtener JWT válido de Supabase; al cerrar, invalidar la sesión local (el JWT en cliente) y cualquier info sensible; redirigir a pantalla apropiada.

- **Publicar Viaje (Conductor):** *"Como conductor, quiero publicar un viaje indicando origen, destino, fecha/hora, asientos y precio, para que pasajeros puedan verlo y reservar."* **Criterios:** Formulario de publicación con validaciones (fechas futuras, precios numéricos, etc.); Al guardar, crear registro en BD vinculado a conductor; viaje visible en búsquedas; solo usuarios autenticados con rol conductor pueden publicar.

- **Buscar y Ver Viajes (Pasajero):** *"Como pasajero, quiero buscar viajes disponibles filtrando por origen, destino y fecha, para encontrar opciones que se ajusten a mi ruta."* **Criterios:** Campos de búsqueda auto-completados (lista de ciudades conocidas, etc., podría almacenarse catálogo); mostrar resultados con info relevante (conductor, hora, precio, plazas libres, rating conductor); si no hay resultados, informar adecuadamente; búsqueda sin login permitido, pero para reservar requerir login.

- **Reservar Asiento (Pasajero):** *"Como pasajero, quiero reservar un asiento en un viaje seleccionado, para asegurar mi lugar en ese viaje."* **Criterios:** Debe estar autenticado; comprobar que el viaje tiene

asientos disponibles; crear registro de reserva (pasajero, viaje, timestamp); si implementamos confirmación, el viaje/usuario conductor recibe notificación y puede aceptar/denegar; si no, la reserva se marca confirmada automáticamente; reducir cupo disponible; evitar doble reservación del mismo asiento por distintas personas (transacción/lock en BD).

- **Notificación de Reserva (Conductor):** *"Como conductor, quiero ser notificado cuando alguien reserve un asiento en mi viaje, para coordinar el encuentro."* **Criterios:** En MVP, esto puede ser una simple notificación in-app (aparece en el panel del conductor que X persona reservó) o un email automático usando algún servicio (Supabase Functions o a futuro). Debe listar datos de contacto del pasajero.

- **Perfil y Reputación:** *"Como usuario, quiero actualizar mi perfil con información personal (foto, teléfono, bio) y ver mi reputación (calificaciones), para generar confianza con otros."* **Criterios:** Pantalla de perfil editable (foto almacenada en Supabase Storage u otro bucket S3, o link); campos como nombre, teléfono, descripción; verificación (quizás marcar manualmente perfiles verificados en admin); visualización de rating promedio y comentarios recibidos tras viajes; para nuevos usuarios sin viajes, indicar que aún no tienen calificaciones.

- **Calificar Usuario tras Viaje:** *"Como pasajero o conductor, al finalizar un viaje quiero poder calificar y dejar una reseña sobre la otra parte, para ayudar a mantener la confianza en la comunidad."* **Criterios:** Solo disponible una vez la fecha del viaje pasó y reserva completada; formulario de calificación (ej: 1 a 5 estrellas) y comentario; una vez enviado se guarda en BD vinculado a los perfiles involucrados; cada usuario puede calificar una vez por viaje; la calificación se refleja en el perfil del usuario calificado (cálculo de promedio, número de reseñas).

(Nota: Estas historias de usuario se plasmarán en especificaciones formales. Según el enfoque SDD, podríamos crear por ejemplo un archivo Markdown por cada funcionalidad principal que incluya: descripción general, historia de usuario, supuestos de negocio, criterios de aceptación (pruebas de aceptación), diseño técnico propuesto – como endpoints, modelos de datos involucrados – y quizás un desglose de tareas técnicas. Estas especificaciones servirán luego como prompt base para generación de código o para guiar el desarrollo manual con mayor certidumbre [7].)

Requerimientos No Funcionales

Además de las funcionalidades, el proyecto debe cumplir con varios **criterios de calidad** comunes en la industria, para asegurar que el producto sea usable y escalable:

- **Seguridad:** Dado que se manejan datos personales y potencialmente pagos, la seguridad es prioritaria. Supabase Auth maneja la autenticación de forma segura (almacenando contraseñas hasheadas, generando JWT). En el backend, implementaremos **verificación de token JWT en cada request protegida** para confirmar la identidad del usuario [5]. Se usará HTTPS para todas las comunicaciones (Vercel y otros servicios lo soportan por defecto). Habilitaremos políticas de **CORS** estrictas en FastAPI, solo permitiendo peticiones desde el dominio de la frontend (por ej. el dominio en Vercel) [8]. También aplicaremos controles de autorización en endpoints sensibles (por ejemplo, un conductor solo puede ver reservas de sus viajes, no de otros). La base de datos habilitará **Row Level Security** (Supabase lo soporta) si se accede directamente, aunque en nuestro diseño principalmente el backend controlará el acceso. Por último, consideraremos validaciones para

prevenir inyecciones SQL (Prisma/Python ORM ya ayuda), sanitización de inputs para prevenir XSS en el frontend, y limitar intentos de login (Supabase podría gestionar parte de esto).

- **Performance y Escalabilidad:** Al iniciar con un MVP, la escala de usuarios es baja, pero debemos diseñar con crecimiento en mente. Next.js nos permite servir una web rápida (SSR/SSG); Tailwind CSS y HeroUI aseguran componentes ligeros. En el backend, FastAPI es muy performante (basado en ASGI/Starlette, puede manejar muchas requests concurrentes). PostgreSQL manejado por Supabase debería soportar la carga inicial; definiremos índices en columnas de búsqueda (ej. índices por origen/destino para consultas de viajes). Se puede configurar caching de consultas frecuentes si fuera necesario en un futuro (Supabase ofrece *PostgREST* caching, pero en nuestro caso el backend custom manejaría las consultas – podemos añadir caching en capa API Gateway si se implementa). Para escalabilidad horizontal, se prevé usar contenedores Docker que permiten desplegar múltiples instancias del backend detrás de un balanceador, y Supabase a su vez escala la BD verticalmente o con read replicas si es necesario. En cuanto a la arquitectura, separar frontend y backend nos ayuda a escalar cada componente independientemente (por ejemplo, Vercel edge network para frontend, y backend desplegado en infraestructura que podamos aumentar en potencia según tráfico) [9].

- **Usabilidad y UX:** La aplicación debe ser intuitiva dado que replicamos un modelo conocido. Emplearemos componentes de **HeroUI** (basado en Tailwind) para tener un diseño moderno y consistente. Se prestará especial atención a flujos clave: registro/login, publicar viaje, búsqueda y reserva, para que requieran la mínima fricción. También consideraremos las condiciones locales: quizás proveer una interfaz liviana que funcione en conexiones lentas (por eso se integrará **PWA**, ver más abajo). Textos claros en español, retroalimentación al usuario (mensajes de éxito/error), y responsive design (muchos usarán la plataforma desde móviles) son requisitos no funcionales de UX. Realizaremos pruebas de usabilidad básicas ad-hoc e iteraremos en mejoras UI durante el desarrollo.

- **Confiabilidad y Disponibilidad:** La plataforma debe estar disponible la mayor parte del tiempo, especialmente en fines de semana o festivos donde podría haber más viajes. Usar servicios en la nube (Vercel, Supabase) nos da alta disponibilidad de base. Configuraremos monitoreos simples – por ejemplo, pings al backend (healthcheck) – y alertas en caso de caídas. Los errores en backend deben manejarse graciosa y registrarse (podemos integrar una herramienta de logging/monitoring como Sentry en el futuro para capturar excepciones). Se planificará también una **estrategia de backup** de la base de datos (Supabase ofrece backups automáticos diarios).

- **Mantenibilidad y Escalabilidad de Desarrollo:** Si bien inicia con un solo desarrollador, es importante escribir código claro y bien estructurado para permitir escalar el equipo a futuro o facilitar contribuciones open-source. El uso de TypeScript en frontend (Next.js) y Python tipado en backend (FastAPI) ayuda a tener tipado estático, reduciendo errores. Se documentarán las partes críticas del código y adoptaremos una arquitectura modular (por ejemplo, en el backend separar routers por contexto: auth, viajes, reservas, perfiles; en el frontend separar componentes UI por páginas y componentes reutilizables). También mantendremos un archivo README o *developer guide* en el repositorio para guiar sobre cómo levantar el proyecto, ejecutar tests, desplegar, etc. La adherencia al enfoque *spec-first* promueve mantenibilidad, ya que las especificaciones actúan como documentación viva del sistema.

- **Portabilidad (Infraestructura como Código):** Empleando Docker para el backend, aseguramos que la aplicación pueda desplegarse en distintos entornos de forma consistente. Definiremos un `Dockerfile` para FastAPI + Prisma, y posiblemente un `docker-compose.yml` que orqueste backend con una instancia de PostgreSQL para desarrollo local. Esto facilita que, si más desarrolladores se unen, puedan replicar el entorno fácilmente. Asimismo, dado que Vercel se encargará del frontend, la configuración de ese despliegue (build commands, etc.) estará en el repositorio. Para backend, si usamos por ejemplo Fly.io, Render o similar, documentaremos los pasos o configuraciones necesarias (puerto, variables de entorno, etc.). Todo esto idealmente formaría parte de la documentación del proyecto.

Arquitectura del Sistema

La solución seguirá una arquitectura web **cliente-servidor** clásica de tres capas, separando claramente el front-end, el back-end y la base de datos [9]. A continuación se describe cada componente y cómo interactúan:

- **Front-End (Cliente Web):** Implementado con **React y Next.js**, será la aplicación que corre en el navegador del usuario. Next.js nos permite renderizado híbrido: páginas estáticas o server-side rendering según convenga (para SEO de páginas públicas, etc.). Todos los elementos de UI se estilizarán con **Tailwind CSS**, aprovechando utilidades de HeroUI para componentes listos (formularios, modales, listas, etc.). El front-end se encargará de las interacciones de usuario: mostrar formularios, listas de viajes, etc., y de llamar a las APIs del backend. Usaremos el **Next.js App Router** (dado que para 2025 Next 13/14 está estable) para organizar las rutas de la aplicación (p.ej. `/register`, `/login`, `/search`, `/trips/[tripId]`, etc.). Muchas de estas páginas requerirán protegerse para solo usuarios autenticados (por ejemplo, la página de publicar viaje, o la vista de mis reservas). Para ello aprovecharemos la integración de Supabase Auth con Next.js: mediante middleware y helpers de `@supabase/auth-helpers-nextjs` podemos redirigir usuarios no autenticados fuera de áreas privadas, y mantener la sesión tanto en cliente como en servidor (SSR) [10] [11]. El front-end almacenará el token JWT de Supabase (probablemente en una cookie HttpOnly o en memoria a través del Auth context) y lo incluirá en las peticiones al backend FastAPI. Además, configuraremos **next-pwa** para convertir la app en una **Progressive Web App**: de ese modo, los usuarios podrán **instalar** la aplicación en sus móviles como si fuera nativa y habrá cierta funcionalidad **offline** (caché de páginas estáticas, assets, y quizás última búsqueda realizada) para mejorar la experiencia en condiciones de poca conectividad. En síntesis, el front debe ser ligero, rápido y seguro. Se desplegará en **Vercel**, aprovechando su CDN global para contenido estático, lo que nos dará bajas latencias en Venezuela también.

- **Back-End (API y Lógica de Negocio):** Implementado con **FastAPI (Python)**, expone un conjunto de **endpoints RESTful** que el front-end consumirá. Esta capa contiene la lógica de negocio central: validación de datos, reglas (e.g., no permitir reservar más asientos que disponibles, etc.), envío de notificaciones, etc. Organizaríamos el código posiblemente en routers (FastAPI routers) como `auth.py`, `rides.py`, `bookings.py`, `users.py` para mantener claridad. Cada ruta estará protegida con autenticación cuando corresponda, usando una dependencia de FastAPI que valide el JWT de Supabase en cada request [5] [6]. Para ello, al iniciar la app, crearemos un cliente de Supabase (usando la clave secreta) para verificar tokens y quizás para otras operaciones si se necesitara [12]. La comunicación con el front será sobre HTTP(S), y dado que estarán en dominios

distintos (ej: frontend en vercel.app y backend en otro host), activaremos **CORS** en FastAPI permitiendo el dominio del front [13] [8].

El backend también interactúa con la **Base de Datos**. Aquí emplearemos **Prisma** como capa de acceso a datos: definiremos nuestro esquema de BD en el archivo `schema.prisma`, que incluirá modelos como User (o tal vez usaremos directamente las tablas de Supabase Auth), Profile, Ride, Booking, Review, etc. Prisma nos permitirá luego generar un cliente Python fuertemente tipado para hacer consultas. Se opta por Prisma sobre ORMs tradicionales de Python (SQLAlchemy) para mantener uniformidad si eventualmente parte del stack se mueve a Node (además Prisma facilita migraciones y tiene un motor de consulta muy optimizado en Rust [14]). Utilizaremos **Prisma Migrate** para gestionar cambios de esquema de forma controlada, generando migraciones versionadas que podemos ejecutar contra la base de datos de Supabase [15]. Durante el desarrollo local, Prisma puede conectarse a una instancia local de PostgreSQL (o directamente a la de Supabase en la nube usando credenciales, aunque se prefiere aislar desarrollo). El backend ejecutará la lógica de negocio sobre estos datos: por ejemplo, al recibir una solicitud POST de reserva, creará el registro en la tabla bookings, decrementará el campo seats disponibles de rides, etc., todo dentro de una transacción para mantener consistencia.

El **despliegue** del backend probablemente se realice mediante **contenedores Docker**. Podemos crear una imagen Docker del servidor FastAPI que incluya el esquema Prisma compilado. En producción, esta imagen se desplegaría en una plataforma como Heroku, Render, Fly.io, AWS (ECS o Lambda) u otro. Como sugerencia, podríamos usar **Railway** o **Fly.io** para hostear gratuitamente (en etapa inicial) el contenedor con FastAPI. Otra opción es usar **Supabase Edge Functions** para lógica pequeña, pero dado que nuestra lógica es más extensa, mantendremos FastAPI. Si optamos por Vercel Serverless Functions, tendríamos que migrar a Node (no queremos eso ahora). Así que seguramente el backend viva separado; por ejemplo, supongamos desplegar en **Railway** (que provee URL pública). En cualquier caso, el front necesita conocer la URL base del API (configurable via variables de entorno).

- **Base de Datos (PostgreSQL en Supabase):** La base de datos relacional almacena toda la información persistente de la plataforma. **Supabase** nos proporciona un PostgreSQL administrado. Aprovecharemos las funcionalidades de Supabase donde posible: por ejemplo, Supabase Auth crea automáticamente tablas para usuarios (auth.users) – podríamos usar esa tabla como base para nuestros usuarios, o replicar la info en una tabla `profiles` extendida con más campos. Supabase también permite almacenar archivos (imágenes de perfil) en su Storage y emitir eventos en tiempo real, que podríamos usar para implementar notificaciones de nuevos viajes o chat (esto último se puede explorar después). Cada migración de Prisma actualiza el esquema de PostgreSQL. Configuraremos en Supabase las *policies* si se fuera a exponer directamente (pero planificamos que todo acceso sea vía nuestro backend, lo que simplifica la lógica de seguridad a nivel de app). Los modelos principales en la BD serán:

- **users / profiles:** para datos de usuario. Posiblemente usemos `auth.users` de Supabase (contiene id UUID, email, timestamp). Crearemos tabla `profiles` con FK al id de Supabase user, incluyendo campos como nombre, foto_url, etc., y tal vez un campo de role (conductor/pasajero o ambos). También guardaremos rating promedio, número de calificaciones.

- **rides:** representa un viaje publicado por un conductor. Campos: id, origen, destino (podrían ser referencias a catálogos de ciudades), fecha_hora_salida (timestamp), duracion_estimada,

precio_por_asiento, asientos_disponibles, detalles_adicionales, conductor_id (FK a user), timestamps creación/actualización.
- **bookings (reservas):** representa la reserva de un pasajero en un ride. Campos: id, ride_id (FK), pasajero_id (FK user), asientos_reservados (por ahora siempre 1 asiento por reserva, pero dejamos campo por flexibilidad), estado (pendiente, confirmada, cancelada), timestamps. Si implementamos pago online, aquí incluiríamos campos de pago (monto, status de pago).
- **reviews:** para calificaciones/reseñas después de los viajes. Campos: id, ride_id, reviewer_id (quién califica), reviewed_id (a quién se califica, puede ser conductor o pasajero), rating (1-5), comentario, timestamp. Alternativamente, podríamos agregar campos de rating directamente en bookings para simplificar (una reserva puede tener rating_conductor y rating_pasajero), pero normalizar en una tabla separada permite más flexibilidad y múltiple reviews (aunque solo se espera una por par viajero-conductor por viaje).
- **messages (opcional):** si implementamos chat interno, una tabla de mensajes con id, ride_id (o booking_id), sender_id, receiver_id, contenido, timestamp.

Estas entidades y sus relaciones serán detalladas en la especificación técnica. En ese documento se incluirán diagramas o descripciones de las relaciones (por ejemplo: un user puede tener muchos rides (conductor), un ride puede tener muchas bookings, cada booking refiere a un pasajero y un ride, etc.). Definir correctamente claves foráneas, índices (por ejemplo, índice compuesto en rides por origen+destino para búsquedas, o en bookings por user para listar histórico) y restricciones (única reserva por usuario por ride, etc.) será parte del diseño de base de datos.

Diagrama de Despliegue e Integración: En resumen, la aplicación se compondrá de varios servicios integrados:

- El **cliente Next.js** (desplegado en Vercel) servirá la interfaz al usuario y se comunicará vía HTTPS con
- el **API FastAPI** (desplegado en un servicio de contenedores) para todas las operaciones de negocio.
- El backend a su vez interactúa con la **BD PostgreSQL de Supabase** (en la nube de Supabase) para almacenar y obtener datos, y con el **servicio de autenticación de Supabase** para validar tokens de usuario.
- Supabase Auth actuará como nuestro proveedor de identidad (gestiona emails/contraseñas y genera JWT).
- No habrá comunicación directa del front con la base de datos; toda la información viaja a través del backend (salvo las operaciones de autenticación inicial que van del front a Supabase Auth endpoints, manejadas por la librería supabase-js en frontend).

Esta separación sigue buenas prácticas de arquitectura (desacoplar frontend, backend y base de datos) permitiendo escalar cada capa independientemente y mantener una seguridad adecuada (la BD no es expuesta públicamente, solo a través del backend) [9]. También facilita el desarrollo concurrente (aunque seamos solo uno, modularizar ayuda).

Metodología de Desarrollo y Gestión del Proyecto

Para gestionar el desarrollo de manera eficiente siendo un solo desarrollador, adoptaremos una combinación de **metodología Ágil** clásica con principios de **Scrum/Kanban**, junto con el enfoque de **Spec-Driven Development (SDD)** potenciado por IA. La industria del software ha hecho de Agile el estándar

dominante para la gestión de proyectos [16], incluso en contextos de startups individuales se pueden aplicar sus prácticas adaptadas. A continuación se detalla cómo organizaremos el trabajo:

- **Planificación y Backlog:** Comenzaremos elaborando un **Product Backlog** que contenga todas las funcionalidades e ideas a implementar, priorizadas según valor para el usuario. Incluso siendo un solo desarrollador, mantener un backlog estructurado ayuda a tener visión global y enfocarse en las tareas importantes primero. Cada entrada del backlog corresponderá a una **historia de usuario** o funcionalidad (muchas de las listadas en la sección de Requerimientos Funcionales). Conforme a Scrum, seleccionaremos un conjunto de esas historias para trabajarlas en un **Sprint** (ejemplo: intervalos de 1 o 2 semanas, según convenga, dado que al ser uno mismo es flexible pero conviene autoimponerse plazos cortos para entregar valor incremental). Durante la planificación de cada sprint definiremos las **tareas técnicas** necesarias para cumplir cada historia y estimaremos el esfuerzo. Llevar un **burndown chart** o al menos anotar el avance ayudará a monitorear el progreso [17]. Aunque prácticas como estimación con story points y medir velocidad son más relevantes en equipos, pueden adaptarse individualmente para autocompromiso. Lo esencial es **entregar iterativamente**: al final de cada sprint, debería haber algún incremento útil del producto (ej.: "al finalizar la Sprint 1, el registro/login y la publicación de viajes están funcionales"). Esto permite validar integraciones paso a paso y ajustar sobre la marcha requisitos que cambien, alineado al principio ágil de adaptabilidad continua.

- **Desarrollo Guiado por Especificaciones (SDD):** Antes de empezar a programar una funcionalidad, dedicaremos tiempo a elaborar su **especificación detallada**. Esta es la pieza central de nuestra metodología: por cada funcionalidad importante crearemos un documento (por ejemplo, archivo Markdown) donde se describa el objetivo de la funcionalidad, su alcance, **criterios de aceptación**, diseño técnico (incluyendo tal vez seudocódigo, definición de endpoints, estructuras de datos, esquemas de BD involucrados) y un desglose de tareas. Esta especificación servirá para pensar el problema con antelación y también como input para las herramientas de IA (el "codex GPT CLI" o agentes que utilicemos podrán leer estas especificaciones y generar código siguiendo esas pautas). Un enfoque estándar que se está formando en la industria es incluso separar diferentes niveles de especificación para cada funcionalidad [18]. Por ejemplo:

  - Un archivo de **especificación de negocio** (qué debe hacer la funcionalidad en términos del usuario y reglas de negocio),
  - otro de **plan técnico** (cómo implementarlo: qué componentes tocar, qué algoritmos o patrones usar),
  - y si es complejo, incluso un listado de **tareas detalladas** para implementarlo en código.

Herramientas recientes como *Kiro (AWS)* o *Spec-Kit (GitHub)* promueven este flujo en 2-3 pasos [19] [18], pero nosotros podemos gestionarlo manualmente con archivos Markdown en el repositorio. El nivel de formalidad lo ajustaremos según la complejidad de cada feature: para algo sencillo (ej. endpoint de listar ciudades) tal vez basta una breve spec; para algo complejo (ej. sistema de reservas con pagos) conviene desglosar en subtareas y varios archivos. Todas estas especificaciones estarán versionadas en Git, de modo que sirven también para historial de decisiones y para futuras incorporaciones al equipo poder entender qué hace el sistema. Además, seguirán actuando como documentación viva. En resumen, **"especificación antes que implementación"** será nuestro mantra: con ello evitamos improvisar en código, reducimos

retrabajo y facilitamos que la IA nos ayude de forma controlada, evitando desviaciones o supuestos ocultos [3].

- **Integración de IA en el Desarrollo:** Dado que queremos alimentar estas especificaciones a un agente GPT (Codex) para asistencia en codificación, debemos estructurar la información claramente. Probablemente utilizaremos un CLI (línea de comandos) donde invocamos a GPT-4/Codex, proporcionándole ya sea el documento completo o secciones relevantes. Por ejemplo, podríamos indicarle: *"Genera el modelo de datos para la entidad Ride según la siguiente especificación..."* usando la parte de la spec donde definimos la tabla de rides. O *"Implementa el endpoint POST /rides según estos criterios de aceptación..."* y suministrar la porción correspondiente de la spec. Para maximizar efectividad, las especificaciones incluirán ejemplos de entradas/salidas esperadas (pseudocódigo de tests) que la IA pueda usar para validar que el código cumple con los criterios. Este proceso es parte del *Spec-Driven Development*: el flujo sería **idea -> especificación -> generación de código -> revisión** [20]. Aún con generación automática, el código producido se revisará manualmente y se ajustará según haga falta (la IA puede ahorrar tiempo, pero el control de calidad sigue en nuestras manos).

- **Control de Versiones (Git) y Colaboración:** El proyecto se alojará en un repositorio **GitHub**. Aunque seamos solo uno, trabajaremos con las buenas prácticas de la industria: ramas para features/bugs, *pull requests* para integrar cambios (posiblemente auto-aprobados pero sirven para documentar contexto de cambios), y mensajes de commit claros. Esto crea un historial ordenado y facilita eventualmente que otros puedan colaborar. Configuraremos protecciones básicas en la rama principal (main) para evitar commits directos accidentales sin pruebas. Cada especificación o tarea grande podría tener su propia rama de implementación. Dado que combinamos con SDD, podríamos incluso subir los archivos `.spec.md` primero en una rama, revisarlos (auto-revisión), y luego en la misma rama o una nueva implementar el código relacionado, manteniendo así trazabilidad entre especificación y código (quizás referenciando en el PR qué spec se está cubriendo). Asimismo, GitHub Issues se pueden utilizar para rastrear tareas o bugs; podemos crear un Issue por cada historia de usuario o tarea técnica, vinculando a la especificación y luego cerrando el issue cuando se completa la implementación. Este estilo de trabajo asegura que nada se pase por alto y que haya referencia cruzada entre planificación (issues), definición (specs) y ejecución (código).

- **Integración Continua y Entrega Continua (CI/CD):** Adoptaremos una estrategia de **Integración Continua** usando **GitHub Actions**. Configuraremos workflows para que en cada push o pull request hacia la rama principal se ejecuten automáticamente los **tests** de la aplicación (tanto del frontend como del backend) [21]. Esto nos permitirá detectar errores de integración de inmediato, manteniendo la base de código siempre en estado desplegable. Por ejemplo, cada vez que implementemos una nueva función, antes de fusionarla a `main`, la acción de CI:

  - Instalará dependencias,
  - Correrá el linters/formatters (podemos usar ESLint/Prettier en front, flake8/black en backend),
  - Ejecutará **pytest** para los tests de backend (posiblemente usando una BD de pruebas o mocks de Supabase),
  - Ejecutará **Playwright** o pruebas de React (por ejemplo, con Jest o Vitest) para frontend.

Si todo pasa verde, entonces podemos hacer merge con confianza. Además, configuraremos CD (Entrega Continua) en la medida de lo posible: dado que el frontend está en Vercel, aprovecharemos su integración

con GitHub – al hacer push a main, Vercel automáticamente construye y despliega la última versión de la app (podemos tener un entorno preview para PRs también). Para el backend, si usamos Railway/Render, también se pueden automatizar despliegues en cada cambio de main (webhooks o GH Actions que construyan la imagen Docker y la suban). Lo ideal es tener un entorno **staging** (pruebas) y **producción**. Por simplicidad, podríamos considerar que la rama `main` despliega a producción directamente una vez validada por CI, pero si quisiéramos ser cautos podríamos tener `main` -> staging, y luego un tag/release para producción. Dado el tamaño inicial, main->prod está bien, con la condición de no pushear código no probado. En cualquier caso, el pipeline de CI nos garantiza calidad en cada cambio. Esta disciplina de CI/CD es considerada una buena práctica estándar hoy en día, permitiendo entregas rápidas y confiables de software [22] [23].

- **Pruebas (QA y TDD):** Vamos a aplicar pruebas automatizadas en distintos niveles:
- **Test unitarios y de lógica (backend):** Usando **pytest**, escribiremos tests para funciones críticas de negocio (por ejemplo, que `crear_reserva()` no permita reservar si no hay cupo, que `calcular_rating_promedio()` devuelva el valor correcto, etc.). También podemos simular peticiones a endpoints con el cliente de Test de FastAPI para verificar respuestas HTTP y lógica end-to-end del API sin levantar servidor real. Siempre que se corrige un bug, añadiremos un test que lo reproduzca para evitar regresiones. Siempre que se añade una nueva función importante, idealmente escribiremos primero su especificación y **criterios de aceptación**, y estos criterios se traducirán en casos de prueba. Esto es similar a Behavior Driven Development: pruebas que reflejan requisitos de negocio.
- **Test de integración (frontend-backend):** Con **Playwright**, automatizaremos escenarios completos como: *"registrar usuario, publicar viaje, buscar viaje, reservar asiento"* para verificar que el flujo entero funciona. Playwright puede ejecutar la app en un navegador headless, interactuar con la UI y validar resultados visibles. Para ello necesitaremos un entorno de pruebas donde frontend y backend estén levantados (podría ser desplegar la app en staging o levantar servicios docker localmente en GH Actions). Estos tests end-to-end serán muy valiosos para detectar cualquier problema en la interacción entre componentes (por ejemplo, un fallo de CORS, o un mal formateo de datos del backend que el front no maneja).
- **Test de componentes (frontend):** Podríamos incluir tests unitarios de componentes React (con Jest/React Testing Library) para verificar que, dado cierto estado o props, la interfaz renderiza lo esperado. No es tan prioritario como los end-to-end en nuestro caso, pero ayudan en componentes complejos (ej: un componente de formulario de publicar viaje: testear que valida campos, etc.).

Adoptaremos una mentalidad de **Desarrollo guiado por pruebas (TDD)** en la medida de lo posible: escribir primero los tests que cubren los criterios de aceptación de una funcionalidad, verlos fallar, y luego implementar el código hasta que pasen. Esto asegura que siempre codificamos con un objetivo claro (hacer pasar el test que representa el requisito) y resulta en un código inherentemente validado. Dado que integraremos IA, podríamos incluso pedirle que genere casos de prueba a partir de la especificación ("escribe pruebas pytest para el caso de reserva sin asientos disponibles según la spec"), lo cual puede acelerar el proceso. En resumen, las pruebas automáticas serán parte integral del ciclo; junto con CI garantizarán que el proyecto avance con calidad y que podamos refactorizar o añadir nuevas características sin miedo a romper lo existente.

- **Herramientas de Gestión y Documentación:** Aparte de GitHub (código, issues, actions), podríamos utilizar herramientas ligeras para organizarnos: por ejemplo, **GitHub Projects** (tableros Kanban) o un tablero en **Trello**/Notion para visualizar el estado de tareas (ToDo / Doing / Done). Esto no es

estricto, pero tener un **tablero Kanban** puede darnos claridad diaria de en qué estamos trabajando y qué queda. En cuanto a documentación adicional, mantendremos actualizada la especificación conforme haya cambios en requisitos. Asimismo, elaboraremos un **Manual de Usuario** básico (aunque al inicio la audiencia seremos nosotros mismos y testers amigos) y un documento de **Plan de Lanzamiento** cuando nos acerquemos a producción (incluirá cómo haremos pruebas beta, recolección de feedback, marketing, etc., aunque eso es más de negocio).

- **Cronograma Tentativo:** Podemos organizar el trabajo en fases:

- **Fase de Configuración Inicial (1 semana):** Configurar el repo, CI, base de datos Supabase, crear proyecto Next.js, proyecto FastAPI, integrar Supabase Auth en ambos lados, probar un flujo mínimo (ej: registro/login funcionando E2E). Escribir especificaciones base para autenticación y set-up.
- **Fase MVP Core (4-6 semanas):** Implementar funcionalidades principales: publicación de viajes, búsqueda y listado, reserva de viaje. Aquí cada sub-función lleva su spec y desarrollo. Al final de esta fase, debería ser posible realizar el flujo completo (sin algunos extras como chat o pagos en línea). Requiere escribir bastante código backend (models, endpoints) y frontend (páginas Next, formularios, etc.). Se realizan pruebas integrales.
- **Fase de Features Complementarias (2-3 semanas):** Agregar calificaciones/reseñas, perfiles de usuario con edición, quizás un sistema básico de mensajería/notifications. También implementar mejoras en UI/UX (pulir estilos, manejo de errores).
- **Fase de Pruebas Finales y Depuración (1-2 semanas):** Testeo intensivo de todo el sistema (QA), corrección de bugs, optimizaciones menores (ej: mejorar performance de alguna query si detectamos lentitud, etc.).
- **Lanzamiento Beta:** Desplegar versión 1.0 en producción, con un grupo controlado de usuarios iniciales. Monitorear métricas (errores, feedback) y planificar iteraciones siguientes según input real.

(Las estimaciones son indicativas; gracias al uso de AI en codificación y a la experiencia previa, quizás algunas tareas se completen más rápido, pero siempre existe incertidumbre).

Desarrollo Tecnológico Específico (Stack)

En esta sección resumimos cómo cada tecnología del stack definido será utilizada concretamente en el proyecto, asegurando que seguimos las **buenas prácticas estándar** para cada una:

- **Next.js (React):** Estructuraremos la aplicación usando la funcionalidad del **App Router** (carpetas `app/` en lugar de `pages/`, ya que es la tendencia actual en Next 13+). Dividiremos la UI en componentes reutilizables (por ejemplo, componentes para *Card de viaje*, *Formulario de búsqueda*, *Formulario de publicar viaje*, etc.) dentro de una carpeta `components/`. Usaremos **SSR** para páginas que muestran datos dinámicos de SEO público, por ejemplo la lista de viajes disponibles podría renderizarse en servidor si quisiéramos indexación (aunque en una app de carpooling puede no ser crucial el SEO, podríamos optar por CSR para simplicidad; decidiremos en la spec de cada página). Next.js facilitará la creación de rutas protegidas: implementaremos middleware que redirija a login si no hay sesión (apoyándonos en `@supabase/auth-helpers-nextjs` como mencionamos antes). También utilizaremos **API Routes** de Next.js *solo si es necesario para alguna integración específica del front*, pero dado que tenemos FastAPI, posiblemente no usemos mucho las API routes de Next (podríamos, por ejemplo, usar una API route para generar un sitemap.xml o algo por el estilo, pero la lógica principal residirá en FastAPI). Configuraremos **next-pwa** plugin para generar el service

worker y manifest de PWA – de modo que la app pueda funcionar offline en modo leído (mostrar páginas ya cargadas o un mensaje "estás offline" de manera amigable) y permitir instalación. A nivel de estado global en React, utilizaremos React Context para la sesión de usuario (posiblemente ya proveído por supabase auth helper) para saber en cualquier componente si el usuario está logueado, sus datos, etc. Para llamadas al backend, probablemente usemos la función `fetch` nativa o alguna librería como SWR/React Query para manejo de datos remotos con cache. Esto último puede ser útil: podríamos usar **React Query** para las consultas de viajes, así tenemos caching de resultados de búsqueda y actualizaciones automáticas si algo cambia. En desarrollo, correremos Next.js con `npm run dev` y en producción se hará build estático (`next build` -> `next start` o servido por Vercel). Mantendremos la configuración de `tsconfig` y `eslint` para seguir buenas prácticas de coding style.

- **Tailwind CSS & HeroUI:** Adoptaremos un diseño mobile-first y limpio. Tailwind nos permitirá iterar rápido en estilo sin salir de JSX, y HeroUI (que al parecer es la continuación de NextUI, una librería UI basada en Tailwind) nos dará componentes ya preparados estilísticamente [24]. Seguiremos la guía de HeroUI para configurar su theming y variaciones en el archivo tailwind.config.js [25]. Así tendremos consistencia en colores, tipografías, etc., posiblemente adaptando un poco a identidiad local (ejemplo: paleta de colores agradable, quizás inspirada en motivos venezolanos sutiles para darle identidad). Nos aseguraremos de revisar la **accesibilidad** de la interfaz: textos alternativos en imágenes, contraste suficiente de colores, y que la navegación sea usable vía teclado y lectores (HeroUI promete accesibilidad por estar basado en React Aria). Al crear las páginas y componentes, usaremos las utilidades de **Flexbox, Grid** de Tailwind para layouts responsivos, evitando CSS crudo. Para iconos, posiblemente usemos **Heroicons** (con Tailwind) o FontAwesome según encaje. En definitiva, el objetivo es tener una interfaz atractiva, moderna pero ligera; gracias a Tailwind evitaremos CSS innecesario y con Purge de Tailwind en producción solo se incluirán los estilos usados, contribuyendo al performance.

- **FastAPI (Python):** En el backend, seguiremos la estructura recomendada por FastAPI: crear instancia de FastAPI, incluir *routers* para distintas secciones, utilizar **Pydantic** models para validar inputs y definir outputs (schema models). Cada endpoint tendrá tipados los modelos de request/response, lo que facilita tanto la validación automática como la generación de documentación interactiva (Swagger UI autogenerado por FastAPI, que podremos habilitar en `/docs` para pruebas manuales de APIs). Implementaremos middlewares necesarios, por ejemplo:

- **CORS** (usando `fastapi.middleware.cors.CORSMiddleware`) para permitir nuestro dominio front [13],
- Autenticación: realizaremos un **Dependency** global que use la clase `HTTPBearer` de FastAPI Security para obtener el token JWT de la cabecera y luego lo valide con la librería `supabase-py` usando `supabase.auth.get_user(token)` [5]. Si no es válido, lanzamos HTTPException 401 [6]. Esta dependencia se aplicará en routers que requieran auth (e.g., `app.include_router(rides_router, prefix="/rides", dependencies=[Depends(verify_jwt)])` para proteger todas las operaciones de rides, o a nivel de cada endpoint).
- Logging: por defecto FastAPI muestra logs de requests. Podemos integrarlo con estructured logging (optional).
- Errores globales: podemos añadir un manejador para excepciones no controladas para loguearlas.

Con respecto a **Prisma Client Python**, seguiremos la documentación oficial para inicializarlo. Básicamente, tras definir `schema.prisma` y correr `prisma generate`, obtendremos un paquete (o module) para interactuar. Tendremos que asegurarnos de ejecutar `prisma migrate deploy` en arranque en producción para aplicar migraciones pendientes, o integrarlo en CI/CD. Durante desarrollo, `prisma migrate dev` ayudará a iterar el esquema. El acceso a la base la encapsularemos quizás en un archivo `db.py` donde instanciamos el Prisma client. Importante: configurar correctamente la URL de conexión a Supabase (proveída en supabase config) y las opciones de pool. Prisma corre un engine separado en Rust que maneja las consultas, por lo que quizás tendremos que tener cuidado de apagarlo al terminar (en FastAPI, usar eventos startup/shutdown para conectar/desconectar a BD si aplica, aunque Prisma maneja pooling internamente).

Para operaciones donde Prisma no encaje (por ejemplo, alguna consulta SQL específica), podríamos usar directamente SQLAlchemy core o hasta las funciones nativas de Supabase (pero en general Prisma debería cubrir la mayoría de CRUD sencillo y queries con filtros). Un beneficio es que Prisma nos permite abstraernos de SQL y trabajar con Python de forma tipo ORM pero generando consultas eficientes.

En cuanto a **estructuración del código**: - Podríamos tener una carpeta `app/` con submódulos: `models/` (Pydantic models), `schemas/` (maybe duplicates of models or directly use Pydantic as schemas), `routers/` (each file for a route group), `services/` (business logic functions, e.g. a booking service function that encapsulates multi-step logic), and `db/` (prisma client). - Mantendremos las funciones lo más simples posible y delegando a helpers para lógica reutilizable. Ej: una función `create_ride(user, data)` en services que haga las validaciones y Prisma calls, y la ruta simplemente llama a esa y maneja la respuesta/código HTTP.

Para pruebas del backend, utilizaremos la capacidad de FastAPI TestClient para simular llamadas a las rutas sin necesidad de desplegar un servidor web real. También podremos usar una **base de datos temporal** (quizá SQLite en memoria) or preferiblemente levantar un contenedor PostgreSQL exclusivamente para tests (pudiendo aplicar migraciones en test para tener schema, y usar transacciones). Esto se integrará en pytest fixtures.

- **Prisma (ORM) y PostgreSQL (Supabase):** Ya cubierto parcialmente arriba, solo recalcar algunas **prácticas**: cada cambio de modelo en `schema.prisma` se acompaña de una migración con nombre descriptivo. Prisma Migrate garantizará esquema sincronizado entre entornos. Tras cada migración que afecte datos (por ej. se añade una columna no nullable), probaremos en Supabase local primero y consideraremos datos por defecto o migración manual de valores. Supabase en modo desarrollo nos permite resetear la base si algo va mal, pero en producción real seremos cuidadosos. También aprovecharemos características de PostgreSQL: **indexes** a través del schema.prisma (Prisma permite declararlos), **constraints** (unique, foreign keys on delete cascade for clean deletions if needed), **views or stored procedures** si en algún caso complejo las necesitamos (Supabase permite añadir funciones SQL; Prisma puede llamarlas vía raw query). Sin embargo, para mantener simplicidad, la mayoría de lógica residirá en Python.

Supabase Auth merece mención: integrando con FastAPI, habrá que configurar la *JWT secret* y *JWKS* if needed. Normalmente Supabase firma los JWT con su secreto (api service role key), y `supabase.auth.get_user(token)` se encarga de validarlo por nosotros (usando la API de Supabase internamente). Es conveniente almacenar la *anon key* y *service role key* en las variables de entorno del backend. El anon key se usa en front para init Supabase client (público). El service role key (secreto)

permitiría incluso hacer operaciones administrativas, pero **no** lo expondremos al frontend, solo backend en entorno seguro. Con Supabase también viene el concepto de *Row Level Security*; inicialmente podemos mantenerlo simple: desactivar RLS en tablas que gestionemos completamente mediante backend, o configurarlas con policies equivalentes a nuestras reglas de negocio (por redundancia en seguridad). Un camino es: la tabla `rides` podría tener RLS que permite solo al owner (conductor) actualizarla/borrarla, etc., y tabla `bookings` permite solo al pasajero o al conductor del ride leerla. No obstante, dado que solo nuestro backend accede, podríamos operar con el service role (superusuario) siempre, y confiar en la lógica de la app para la seguridad. Este detalle se definirá en la spec de seguridad.

- **Supabase (Servicios adicionales):** Además del Postgres y Auth, Supabase provee:
- **Storage:** lo usaremos para almacenar archivos como fotos de perfil, de vehículo, etc., en lugar de guardar binarios en la BD. Gestionado mediante supabase-js del front (subir imagen) y reglas de acceso (por ejemplo, hacerlas públicas o con URL firmadas).
- **Edge Functions:** pequeñas funciones serverless en Node/Denno. Quizás no usemos por tener FastAPI.

- **Realtime:** supabase puede emitir eventos realtime en tablas. Podríamos subscribir desde front para, ej., notificar a un conductor en tiempo real cuando recibe una reserva (sin tener que hacer polling). Esto implicaría habilitar Realtime en la tabla bookings y usar supabase-js on client para escuchar. Lo consideramos opcional para MVP (podemos con simple refresh o mediante WS más adelante), pero es bueno saber que existe.

- **Analytics:** supabase no da mucho analytics, pero podríamos instrumentar Google Analytics en frontend to track usage or some custom events.

- **Vercel (Despliegue Frontend):** Configuraremos el proyecto en Vercel vinculando el repo GitHub. Aprovecharemos variables de entorno en Vercel (por ejemplo la URL del backend API, los keys de supabase público, etc.). Activaremos previews para ramas/PR para poder ver cambios antes de merge. Vercel se encargará de CDN y escalado del front, lo cual es estándar. Debemos verificar que las rutas dinámicas funcionen bien (Next13's routing). También, en Vercel podemos usar su soporte de **redirects/rewrites** si quisiéramos proxy requests a backend, pero probablemente no, mejor el front llame directo al backend URL. Por último, monitorizaremos el tamaño del bundle en build – Next.js y Webpack suelen optimizarlo pero hay que evitar sobrecargar de librerías.

- **Docker:** Crearemos un Dockerfile para el backend. Posiblemente basado en `python:3.11-slim`. Incluirá instalar dependencias (`pip install -r requirements.txt`), instalar prisma (via `pip install prisma-client-py` and have the prisma engine binary accessible – prisma py may handle downloading engine). Montaremos el código, correr `prisma migrate deploy` and `uvicorn main:app`. Para desarrollo, un docker-compose puede incluir un servicio `db` (postgres) y `backend` para facilitar arrancar todo con un comando. Para el frontend, podríamos dockerizarlo también (FROM node:..., build, etc.) pero dado Vercel se encarga, no es prioritario. Sí podríamos tener un docker de front para caso de querer desplegar fuera de Vercel en algún momento.

- **GitHub Actions (CI):** Ya cubierto anteriormente, pero técnicamente tendremos archivos YAML en `.github/workflows/`. Uno para CI: triggers en PR y push a main. Pasos: checkout, setup Node (ci for front), setup Python, maybe setup Postgres service (GH Actions soporta servicios docker, podemos añadir uno postgres:13), luego `pip install` y `npm install`, luego quizás levantar

backend (uvicorn) y front (next dev) en background for integration tests (o ejecutar tests separadamente). Veremos la complejidad; quizá más sencillo correr los tests de backend aislados (usando an in-memory or test DB) y los tests de front con mocking backend or pointing to a staged environment. Por simplicidad, podríamos en CI solo correr tests unitarios de backend y front por separado, y dejar los e2e (Playwright) para correr local antes de deploy o en un pipeline manual, porque requiriría orquestar front+back juntos. Sin embargo, Playwright con Testing Library de Supabase se puede apuntar a un entorno staging para pruebas de humo post-deploy. Se puede configurar un Action para deploy también (aunque Vercel ya se integra solo, podríamos no necesitar manual). Otro workflow a configurar: *Linting* (ejecutar eslint/black) y *Docker build* (validar que la imagen se construye correctamente). Con los resultados de CI integrados en cada PR, mantendremos la calidad continuamente [26] [27].

- **Playwright (E2E tests):** Prepararemos scripts de prueba E2E tras que la app esté desplegada en un entorno de prueba. Playwright nos permitirá simular un browser; definiremos casos como: *"Registro -> Publicar Viaje -> Logout -> Login como pasajero -> Buscar -> Reservar -> Logout -> Login conductor -> Ver reserva"*, etc., para cubrir el journey principal. Estas pruebas las podemos ejecutar localmente y en CI (aunque en CI suele requerir configurar Xvfb or usar la acción oficial de Playwright que instala browsers). Playwright genera bonitos reportes y hasta videos de los tests, útiles para depurar. Nos aseguraremos de tener datos de prueba aislados (quizás creando cuentas especiales "testuser1", "testdriver1" con data). Podríamos automatizar que en CI el backend arranque en modo test con una base de datos volátil, y correr Playwright contra `localhost` del action runner. Esto es un nivel de complejidad avanzada pero factible. A falta de eso, podemos al menos usar Playwright contra un entorno staging desplegado. En cualquier caso, la meta es que antes de cada release, pasen todos los tests E2E, garantizando que funcionalidades críticas no se rompan.

- **pytest (tests backend):** Como mencionado, escribiremos tests para lógica interna. Utilizaremos fixtures de pytest para inicializar una base de datos temporal. Quizás usemos un enfoque: antes de tests, aplicar migraciones en una BD SQLite o un schema de Postgres en memoria; luego usar transacciones para aislar. También podemos mokear el supabase.auth si no queremos depender de ext API en tests (e.g., stub de get_user que devuelve un objeto usuario válido). Pytest integrará bien con CI ya que su salida puede convertirse a JUnit XML para ver resultados.

En conjunto, este stack tecnológico está alineado con las tendencias modernas para aplicaciones web: **React+Next.js en el front** proporciona gran UX, **FastAPI** ofrece rendimiento de backend y facilidad de escritura en Python, **Prisma+Postgres** asegura una capa de datos robusta y tipada, y servicios cloud como **Supabase** y **Vercel** simplifican la infraestructura. Todo esto orquestado con **contenedores** y **CI/CD** nos pone a la par de estándares de la industria en cuanto a entrega de software.

Consideraciones Finales y Siguientes Pasos

Con este plan, hemos cubierto **“todo lo necesario”** para entender el proyecto en su totalidad – desde qué hace y cómo funciona la plataforma BlaBlaCar adaptada, hasta cómo vamos a gestionarla y construirla técnicamente. Para recapitular los puntos clave:

- Adoptamos una metodología **Agile** porque es el estándar en la industria para gestionar proyectos de software de manera flexible y efectiva [16]. Incluso siendo un solo desarrollador, aplicaremos sus principios (entregas incrementales, adaptación al cambio, enfoque en valor al usuario) mediante un

backlog, sprints cortos y retroalimentación continua. Esto asegurará que avancemos de forma controlada y podamos responder rápidamente a nuevos requerimientos o descubrimiento de problemas.

- Complementariamente, utilizaremos **Spec-Driven Development (SDD)** como enfoque de vanguardia para maximizar la claridad y la eficiencia, especialmente dado que planeamos usar asistencia de IA. La especificación detallada de cada funcionalidad será el pilar: define *qué* debe hacer el sistema antes de escribir *cómo* hacerlo en código [3]. Esto alineará implementación con objetivos y permitirá alimentar al agente GPT con información completa para generar código relevante, reduciendo iteraciones de "prueba y error" típicas. Al centralizar la spec, mantenemos trazabilidad y minimizamos ambigüedades en el desarrollo [4].

- El análisis de las **funcionalidades** nos dio un entendimiento profundo de los requisitos del sistema tipo BlaBlaCar: desde registro/login seguro, pasando por el núcleo de publicar y reservar viajes, hasta elementos de reputación y comunicación. Cada uno de estos fue descompuesto en historias y criterios que guiarán su posterior implementación. Asegurar estos fundamentos funcionales es crítico para el éxito del producto, pues la propuesta de valor se sostiene en hacer fácil y confiable compartir viajes entre extraños, lo cual logramos solo si la aplicación implementa bien esas características (buen emparejamiento de viajes, confianza mediante calificaciones, etc.).

- La **arquitectura** propuesta sigue un modelo robusto y escalable, separando responsabilidades: un front-end React responsivo y disponible, un back-end con lógica clara y API bien definida, y un almacenamiento central consistente (PostgreSQL). Esto es coherente con arquitecturas de referencia de aplicaciones web modernas, garantizando mantenibilidad y capacidad de escalar componentes por separado (por ejemplo, si la carga de solicitudes crece, podemos escalar horizontamente instancias de FastAPI sin tocar el front, etc.). Hemos considerado incluso integración de servicios específicos de Supabase para agregar valor (ej: realtime, storage), que podemos introducir conforme crezca la necesidad.

- El **stack tecnológico** seleccionado aprovecha herramientas maduras y eficientes: Next.js para la experiencia de usuario, FastAPI por su rendimiento y facilidad con Python, Prisma para un manejo seguro de base de datos, entre otros. La integración de estos componentes se basará en prácticas recomendadas (ej: verificación JWT en backend [5] [6], CI/CD pipelines [21], etc.) tal como respaldado por las referencias. Cada decisión (como usar Supabase Auth en vez de construir nuestra auth, o usar Vercel para deploy) busca reducir carga de desarrollo en áreas que no son el diferenciador del producto, de forma que podamos enfocarnos en la lógica de negocio de carpooling.

- Hemos delineado un **plan de pruebas y calidad** riguroso: test unitarios, integración, end-to-end, todo automatizado en la medida de lo posible. Esto no solo dará confianza en el producto resultante, sino que también nos permitirá refactorizar o extender funcionalidades sin temor a romper lo existente (los tests actuarán como red de seguridad). Para un solo desarrollador, invertir en automatización de pruebas es invertir en velocidad futura, pues atrapa errores temprano que de otro modo consumirían mucho tiempo manual depurando.

- En términos de **gestión de proyecto**, la disciplina de usar Git/GitHub, gestionar tareas, documentar especificaciones y usar CI/CD nos pone en la senda de un desarrollo profesional y sostenible.

Aunque pudiera parecer overhead para un proyecto unipersonal, en realidad estas prácticas minimizan el riesgo de acumulación de *deuda técnica* y facilitan la incorporación de posibles colaboradores más adelante. Adicionalmente, tener un pipeline de despliegue automático nos acerca a la filosofía "deploy early, deploy often", que es beneficiosa para recopilar feedback rápido de usuarios reales.

Con toda esta preparación, el siguiente paso concreto sería comenzar a elaborar las **especificaciones iniciales** (por ejemplo, una especificación general del sistema, y luego specs particulares para Autenticación, Publicar Viaje, Buscar/Reservar, etc., siguiendo posiblemente el esquema de 2 pasos: spec negocio + plan técnico [28] [18]). Estas especificaciones se alimentarán al agente GPT-CLI para asistir en la generación de código base. Por supuesto, tras generar código, habrá un ciclo de revisión y mejora (tanto manual como potencialmente con la ayuda de la IA refinando prompts). Continuaremos iterando funcionalidad por funcionalidad hasta completar el MVP.

En conclusión, este documento sirve como *north star* tanto para el desarrollador humano como para el agente de IA: define **qué es el proyecto, cómo está estructurado, y bajo qué estándares se desarrollará**. Con ello, tanto "mi agente" (el modelo GPT) como yo tenemos una visión compartida y detallada del camino a seguir, lo que nos permitirá colaborar eficazmente en la construcción de la aplicación BlaBlaCar Venezuela y aumentar las probabilidades de un resultado exitoso. ¡Manos a la obra!

Fuentes: Hemos referenciado buenas prácticas de la industria y especificaciones conocidas durante esta planificación, incluyendo documentación sobre desarrollo guiado por especificaciones [3] [4], características de BlaBlaCar y aplicaciones de carpooling [1] [2], metodologías ágiles aplicadas incluso a proyectos unipersonales [16] [17], integración de nuestra arquitectura propuesta (frontend en Vercel comunicándose con backend FastAPI mediante API segura) [8] [9], y lineamientos técnicos para autenticar con Supabase y asegurar nuestros endpoints [5] [6], así como la importancia de CI/CD con herramientas modernas [21]. Estas referencias respaldan las decisiones tomadas y sirven como consulta para implementar cada aspecto conforme avancemos. Hemos consolidado todo este conocimiento aquí de forma técnica y estructurada, listos para convertirlo en realidad a través del código. [3] [4] [16] [1] [2] [17] [8] [9] [5] [6] [21]

---

[1] [2] Blablacar Clone App Development: Features, Cost & Tech Stack
https://www.imgglobalinfotech.com/blog/build-an-app-like-blablacar

[3] [7] [18] [19] [20] [28] Desarrollo Guiado por Especificaciones - SDD: La IA genera el código a partir de especificaciones
https://aicode.academy/blog/es/spec-driven-development/

[4] GitHub - tobiasfacello/spec-driven-development: Este repositorio reúne guías, ejemplos y buenas prácticas para aplicar Spec Driven Development (SDD) en proyectos de software.
https://github.com/tobiasfacello/spec-driven-development

[5] [6] [10] [11] [12] [13] Implementing Supabase Authentication with Next.js and FastAPI | by Ojas Kapre | Medium
https://medium.com/@ojasskapre/implementing-supabase-authentication-with-next-js-and-fastapi-5656881f449b

[8] [14] Deploy fastAPI with Prisma ORM on Vercel : r/FastAPI
https://www.reddit.com/r/FastAPI/comments/1fh6prp/deploy_fastapi_with_prisma_orm_on_vercel/

9 Web Application Architecture: Front-end, Middleware and Back-end - DEV Community
https://dev.to/techelopment/web-application-architecture-front-end-middleware-and-back-end-2ld7

15 Prisma | Works With Supabase
https://supabase.com/partners/integrations/prisma

16 22 23 Agile Project Management: Best Practices and Methodologies
https://www.altexsoft.com/whitepapers/agile-project-management-best-practices-and-methodologies/

17 Agile for the Solo Developer - Software Engineering Stack Exchange
https://softwareengineering.stackexchange.com/questions/220/agile-for-the-solo-developer

21 26 27 Continuous integration - GitHub Docs
https://docs.github.com/en/actions/get-started/continuous-integration

24 Introduction | HeroUI (Previously NextUI)
https://www.heroui.com/docs/guide/introduction

25 How to use tailwind v4 with Hero UI #16369 - GitHub
https://github.com/tailwindlabs/tailwindcss/discussions/16369