ChatGPT

Delicia – Plataforma Inteligente de Pedidos para Restaurantes

Resumen Ejecutivo

Delicia (nombre temporal inspirado en la palabra *"delicia"* para evocar sabor y destacando "IA" al final para *Inteligencia Artificial*) es una plataforma inteligente de gestión de pedidos dirigida a negocios de comida rápida y restaurantes de cualquier tamaño, desde pequeños puestos hasta grandes cadenas. El sistema centralizará el menú y la toma de pedidos, asistido por una **IA conversacional** integrada que mejora la experiencia del cliente al poder responder preguntas sobre platillos, tomar pedidos mediante chat y guiar al usuario de forma natural.

La plataforma busca **optimizar la eficiencia operativa**: los pedidos de clientes se reflejan en tiempo real en un *Modo Cocina* para los cocineros, los cajeros pueden gestionar pagos y entregas, y los administradores cuentan con un **dashboard visual** para monitorear ventas y operaciones. Todo esto se desarrollará como un **monolito modular** con NestJS en el backend y React en el frontend, usando tecnologías modernas (PostgreSQL, Prisma, Redis, WebSockets, Tailwind, etc.) para asegurar escalabilidad y mantenimiento a largo plazo.

Alcance: El PRD cubre los requerimientos funcionales y técnicos de la plataforma de pedidos, la integración de la IA conversacional con Model Context Protocol, la arquitectura modular del sistema, los roles de usuario y sus flujos, consideraciones de UI/UX (incluyendo microinteracciones y estilo dominicano moderno), y ejemplos de implementación en código. **Quedan fuera de alcance** en esta fase las funcionalidades de promociones inteligentes y campañas de marketing automatizadas, las cuales se podrán incorporar en el futuro una vez establecida la base del sistema.

Objetivos del Producto

- **Mejorar la Experiencia de Pedido:** Permitir a los clientes ordenar comida de forma intuitiva, ya sea navegando el menú tradicionalmente o interactuando con una **IA asistente** que entienda preguntas en lenguaje natural y sugiera opciones. Esto reduce tiempos de espera y errores en pedidos, brindando un trato cercano y eficiente al estilo dominicano.
- **Unificar Operaciones para Todo Tamaño de Negocio:** Ofrecer una solución única que funcione tanto para un puesto de comida rápida como para un restaurante grande. La plataforma será configurable y modular, pudiendo activarse o desactivarse módulos según necesidades (por ejemplo, un puesto pequeño quizá use solo la toma de pedidos básica e interfaz de cocina, mientras un restaurante grande use todos los roles y el dashboard completo).
- **Eficiencia en Tiempo Real:** Implementar capacidades en tiempo real mediante **WebSockets** para notificar instantáneamente nuevos pedidos y cambios de estado. El *Modo Cocina* mostrará órdenes al momento que son ingresadas, y los clientes podrán recibir actualizaciones de su pedido (por ejemplo, "en preparación", "listo para recoger"). Esto mejorará la coordinación entre cocina y caja, reduciendo esperas.
- **Arquitectura Robusta y Escalable:** Construir el sistema con un **diseño modular limpio**, siguiendo principios SOLID y arquitectura hexagonal para mantener bajo acoplamiento. NestJS permite un código mantenible y escalable gracias a su diseño modular y soporte de inyección de

dependencias. Se utilizará **Prisma ORM** por su excelente integración con TypeScript y garantizando consultas seguras y eficientes en PostgreSQL. La inclusión de **Redis** permitirá escalar la app en múltiples instancias sin perder sincronización de eventos en tiempo real (usando un mecanismo Pub/Sub para compartir eventos entre instancias de servidor).

- **Interfaz Atractiva y Familiar:** Desarrollar un frontend con **React + Vite** que ofrezca una experiencia rápida y dinámica. Utilizaremos **Tailwind CSS** y **ShadCN UI** para crear componentes visualmente atractivos y coherentes, incorporando *microinteracciones* (animaciones sutiles al pulsar botones, transiciones al abrir menús, notificaciones suaves cuando llega un pedido nuevo, etc.). La estética de la UI tomará **inspiración moderna dominicana**, usando colores vibrantes y motivos sutiles de la cultura (por ejemplo, paleta tropical, ilustraciones de comida local estilizada) para dar a la plataforma una identidad gráfica distintiva y cercana al usuario local.

Roles de Usuario y Actores

La plataforma manejará **cuatro roles principales**, cada uno con su interfaz y funcionalidades adaptadas:

- **Cliente (Usuario Final):** Persona que realiza el pedido de comida. Puede navegar el menú por categorías, buscar platos específicos y hacer un pedido. Alternativamente, puede entablar una **conversación con la IA asistente** para pedir recomendaciones ("¿Qué me recomiendas con pollo?") o armar su orden por chat. El cliente puede personalizar artículos (ej. elegir ingredientes extra o excluir algo) y finalmente confirmar su pedido. Debe poder ver el estado de su pedido en tiempo real (ej. preparado, en camino si hubiera delivery, listo para recoger, etc.). Si el pedido no se paga por adelantado, recibirá un código o número para pagar en caja.
- **Cocinero (Staff de Cocina):** Usuario encargado de preparar los pedidos. Accede al **Modo Cocina**, una vista en tiempo real que muestra la cola de pedidos entrantes con detalles relevantes: número de orden, items solicitados con sus personalizaciones, tiempos de orden. Esta interfaz se diseñará para pantallas grandes o tablets en la cocina, con actualizaciones automáticas vía WebSocket. El cocinero puede cambiar el estado de cada pedido (por ejemplo, marcar como "En preparación" y luego "Listo") para informar al sistema. La interfaz de cocina enfatiza la **legibilidad** (texto grande, contraste alto) y notificará claramente nuevos pedidos (sonido o resaltado visual).
- **Cajero (Staff de Caja):** Usuario que administra pagos y entregas. En un entorno de mostrador, el cajero verá los pedidos pendientes de pago o entrega. Puede buscar un pedido por código/ID, ver el total a cobrar y marcarlo como pagado. También puede confirmar la entrega del pedido al cliente. La interfaz del cajero será sencilla, listando órdenes recientes y sus estados (pagado/pendiente, entregado/pendiente). Adicionalmente, el cajero podría tener la capacidad de crear pedidos manualmente en nombre de clientes que ordenan en persona, usando la misma interfaz de cliente pero rápida.
- **Administrador (Gerencia):** Usuario encargado de la administración del sistema. Tendrá acceso a un **Dashboard administrativo** con métricas en tiempo real (ventas del día, platos más vendidos, tiempo promedio de preparación, etc.) presentadas mediante gráficas llamativas. El administrador puede gestionar el **menú** (CRUD de categorías y productos, incluyendo subir fotos de platos, precios, descripciones), gestionar usuarios del sistema (crear cuentas de cajeros, cocineros, etc., con roles y permisos), y revisar el historial de pedidos. También podrá configurar parámetros generales (horarios de servicio, impuestos, etc.). La prioridad para este rol es la **visualización clara de datos** y controles de gestión fáciles de usar, manteniendo la estética moderna de la plataforma.

Funcionalidades Clave por Módulo

A continuación se detallan las principales capacidades del sistema, organizadas por áreas funcionales:

- **1. Módulo de Menú y Pedidos (Frontend & Backend):**
- **Navegación de Menú:** Presentación del menú con categorías (ej. Entrantes, Platos fuertes, Bebidas, Postres). Los clientes pueden explorar la carta, ver detalles de cada platillo (descripción, precio, foto) y añadir al carrito. El menú se cargará desde el backend (vía API REST) y puede almacenarse temporalmente en el frontend para rápida navegación, con actualizaciones en tiempo real si el admin cambia algo (por ejemplo, desactivar un platillo agotado).
- **Búsqueda y Filtros:** Barra de búsqueda para encontrar platos por nombre o ingredientes, con filtrado (ej. vegetariano, picante, etc.).
- **Pedido Convencional:** Permitir al cliente armar su pedido seleccionando items del menú, especificando cantidades y variaciones (ej. "sin cebolla", "extra queso"). El **carrito de compra** mostrará el resumen de la orden antes de confirmar. Al confirmar, se envía la orden al backend mediante una API POST (`/orders`). Se implementará validación de stock o disponibilidad al enviar la orden.
- **Asistente de Pedidos con IA:** Integración de un **chatbot** en la interfaz de cliente donde el usuario puede escribir (o potencialmente hablar) de manera conversacional para hacer su pedido. La IA puede entender peticiones en lenguaje natural – por ejemplo, *"Quiero una hamburguesa con todo y una Coca-Cola"* – consultar internamente el menú para identificar productos mencionados, hacer preguntas de aclaración si falta info (*"¿Desea la hamburguesa término medio?"*), y agregar items al pedido virtual. La conversación fluye hasta que el usuario confirma el pedido. Técnicamente, esto se logra exponiendo funciones del backend (buscar items, crear pedido) como **herramientas** que la IA puede invocar vía MCP (ver sección de Integración de IA).

- **Confirmación de Pedido:** Una vez armado el pedido (ya sea vía carrito manual o chat IA), el cliente confirma y el sistema genera un número de pedido. Si aplica pago en línea (no cubierto en esta fase, posiblemente se paga al recoger), se indicará el método. El pedido confirmado dispara notificaciones: aparece en la vista de cocina y caja en tiempo real. El cliente recibe una confirmación (en la UI del cliente o chat).

- **2. Módulo de Cocina (Pedidos en Tiempo Real):**

- **Cola de Pedidos Activos:** Interfaz de cocina mostrando una lista de pedidos pendientes de preparación. Cada pedido se presenta en una tarjeta o fila con código, hora de entrada, listado de platos y personalizaciones. Nuevos pedidos **aparecen automáticamente** sin refrescar la página, gracias a la suscripción a eventos WebSocket. Es decir, el backend emitirá un evento `new_order` cuando un pedido ingresa, y los clientes (navegadores en cocina) lo recibirán instantáneamente para añadirlo a la lista. (En NestJS esto se soporta mediante un *WebSocket Gateway* usando Socket.io, junto con Redis para escalar a múltiples instancias).
- **Actualización de Estado:** El cocinero puede interactuar con cada pedido para cambiar su estado. Ejemplo: botón "Marcar como En Preparación" y luego "Marcar como Listo". Estas acciones envían eventos al backend (vía WebSocket o llamadas API) que actualizan la base de datos y notifica a otras interfaces interesadas. Por ejemplo, al marcar "Listo", el sistema notifica al cajero (su pantalla muestra que X pedido está listo para entregar) y opcionalmente al cliente (si está viendo estado desde su dispositivo).
- **Modo Pantalla Completa y Sonidos:** El modo cocina operará idealmente en pantalla completa para maximizar área utilizable. Se incluirán **indicadores audiovisuales**: p. ej., un sonido o

vibración en la interfaz al llegar un nuevo pedido, y resaltado en amarillo por unos segundos. Asimismo, si un pedido lleva demasiado tiempo en cola, podría resaltarse (indicador de demora).

- **Gestión de Prioridad:** Posibilidad de que el cocinero o administrador reordenen la cola (por ejemplo, priorizar un pedido atrasado). Aunque inicialmente el orden es según llegada, el sistema debe ser capaz de soportar re-priorización manual o automática (esta última podría ser una futura "inteligencia" no incluida aún).

- **3. Módulo de Caja (Gestión de Pagos y Entregas):**

- **Listado de Pedidos Pendientes de Pago:** El cajero verá una lista de pedidos que requieren cobro (por ejemplo, pedidos realizados in situ para pagar en efectivo). Cada entrada mostrará el código de pedido, nombre del cliente (si se obtuvo), total a pagar y estado (esperando pago, pagado). El cajero puede seleccionar un pedido, ver los detalles y registrar el pago (p. ej. marcando método de pago: efectivo, tarjeta). Al registrar el pago, el estado pasa a "Pagado", y podría notificar a cocina que ya está autorizado (en caso de que se espere confirmación de pago para empezar a preparar, aunque normalmente se prepara de inmediato).
- **Gestión de Entregas:** Para pedidos de tipo para recoger en mostrador, el cajero ve cuáles están "Listos para entregar" (es decir, cocina ya los marcó como listos). Puede llamar al cliente (p. ej. por número de pedido o nombre) y al entregar físicamente la comida, marca el pedido como "Entregado/Completado" en el sistema. Esto cerrará el ciclo del pedido.
- **Generación de Tickets o Recibos:** Posibilidad de generar un recibo impreso o digital. En esta fase se puede simplemente mostrar en pantalla para impresión del navegador. (Integraciones con impresoras térmicas específicas podrían considerarse más adelante).

- **Pedidos In-situ por Cajero:** El cajero debe poder crear pedidos rápidos en la situación de que un cliente llegue y ordene verbalmente. Para esto podría utilizar una versión simplificada de la interfaz de pedidos del cliente (sin chat, solo seleccionando items rápidamente) y confirmando para enviarlo a cocina, marcándolo como pagado directamente si cobra al instante. Esto permite usar la plataforma tanto para pedidos autogestionados por el cliente como para pedidos tomados por personal.

- **4. Módulo de Administración y Dashboard:**

- **Dashboard Analítico:** Pantalla principal para administradores con gráficos y KPIs. Por ejemplo, gráfico de barras o pastel con ventas por categoría de producto, número de pedidos en el día vs días anteriores, horas pico de pedidos, etc. Se pueden utilizar librerías como Chart.js o Recharts, integradas con React, para mostrar datos dinámicamente. La arquitectura del dashboard consumirá endpoints del backend (posiblemente vía NestJS con controladores para métricas agregadas). La actualización de ciertas métricas podría ser en vivo (usando WebSockets para contar pedidos del día en tiempo real). Visualmente, este dashboard tendrá un estilo **"moderno limpio"** con toques de identidad dominicana (colores, tipografías amigables) pero manteniendo la claridad.
- **Gestión de Menú:** Interfaz CRUD para administrar categorías e ítems del menú. El admin puede: crear nuevas categorías (nombre, descripción), añadir platos con sus detalles (nombre, ingredientes, precio, carga de imagen). Se validará que todos los campos obligatorios estén completos y se dará feedback de éxito o error con microinteracciones (ej. un modal o toast "Plato guardado exitosamente"). Si un plato se desactiva (ej. ingrediente agotado), al guardarlo como "no disponible", el cambio se transmite a los clientes en tiempo real (por ejemplo, removiéndolo u oscureciéndolo en el menú de la app cliente). **Ejemplo de flujo:** el admin desde el dashboard

quita el producto “Batido de chinola” del menú; al confirmar, el backend emite un evento via WebSocket que hace que cualquier cliente con el menú abierto lo oculte o marque agotado.

- **Gestión de Usuarios y Roles:** El administrador puede crear cuentas para el personal (cajeros, cocineros) asignándoles roles. Podrá restablecer contraseñas, y desactivar/activar usuarios. La seguridad es importante: solo el admin puede manejar roles, y a su vez ciertos datos sensibles (como finanzas) podrían estar ocultos a roles no admin.
- **Configuración General:** Opciones de negocio como horario de apertura/cierre (que podría usarse para que la IA y la app sepan si están aceptando pedidos), impuestos o cargos de servicio, y potencialmente traducciones de menú para modo multilenguaje (futuro). Estas configuraciones se almacenarán en la base de datos y el frontend las consulta para comportamientos correspondientes (por ej., no aceptar pedidos fuera de horario).

(Nota: En esta fase inicial, no se incluyen funcionalidades de promociones inteligentes ni campañas de marketing automatizado. Tampoco se aborda la integración con servicios de delivery externos. Estas características se dejan como ampliaciones futuras fuera del alcance actual.)

Flujos de Usuario Principales

A continuación se describen los flujos de trabajo más importantes desde la perspectiva de los distintos roles, resaltando cómo interactúan con el sistema:

1. Flujo de Pedido de Cliente (con Asistente IA integrado)

1. **Inicio – Exploración del Menú:** El cliente ingresa a la interfaz (vía app web React responsiva). Si es un kiosko en local, puede haber un modo de pantalla completa. El cliente ve la página principal con categorías destacadas o promociones (p.ej. “Combo del Día”), puede hacer scroll o clic en categorías para ver opciones. Alternativamente, puede abrir el **chat de la IA** (un botón tipo “¿Necesitas ayuda? Pregúntale a nuestro asistente”).
2. **Consulta Conversacional (Opcional):** Si usa el chat, el cliente puede escribir algo como “Tengo mucha hambre, ¿qué me recomiendas?” o “Quiero un sandwich y un jugo de piña”. La IA (basada en MCP) procesa la solicitud: internamente llama a herramientas para obtener elementos del menú que coinciden (por ejemplo, busca “sandwich” en la base de datos) y responde con un mensaje amable. Puede decir: “Te sugiero nuestro **Sandwich Especial** con un jugo natural de piña. ¿Te gustaría añadirlo al pedido?” El cliente puede continuar la conversación (ej. “¿Qué trae el sandwich?” – la IA describirá ingredientes; “Sí, añádelo” – la IA registrará el pedido). Todo este flujo se mantiene natural, la IA recuerda el contexto (por ejemplo, sabe que el cliente quiere jugo de piña y sandwich).
3. **Construcción del Pedido:** Ya sea vía chat o manual, el cliente selecciona los productos deseados. En la UI tradicional, añade varios items al carrito. Puede modificar cantidades, quitar algo, o editar (p. ej. “sin azúcar” en el jugo, ingresando notas o seleccionando opciones predefinidas). La app valida en frontend también (por ejemplo, no permitir cantidades negativas, etc.).
4. **Confirmación y Envío:** El cliente revisa el resumen de su pedido en la sección “Tu Orden”. Ve el total calculado (con impuestos si aplican). Si todo está correcto, pulsa “Confirmar Pedido”. En caso de usar IA, la IA puede decir “Voy a confirmar tu pedido de X, total X dinero. ¿Confirmas?” y al decir “sí” procede igual. La aplicación envía la orden al backend (HTTP POST `/orders` con los datos del pedido). Se muestra al usuario una confirmación (pantalla de agradecimiento con número de pedido y recordatorio de pago si debe pagar en caja).
5. **Notificaciones Inmediatas:** Al recibirse el pedido en backend, este crea registros en la base de datos (orden y detalles). En paralelo, el servidor emite eventos en **Tiempo Real**: uno para la interfaz de cocina (“nuevo pedido #123”), otro para la interfaz de caja (si el pago está pendiente,

"pedido #123 esperando pago"). El cliente podría recibir un mensaje en la misma página o chat "¡Gracias! Tu pedido #123 está en preparación".

6. **Espera y Seguimiento:** El cliente espera su comida. Si está en un local y pagará en caja, se dirige a pagar mencionando su número de pedido. El cajero lo marca pagado en el sistema. Mientras, en cocina se prepara la orden. Si el cliente permanece en la página, podría ver actualizaciones: por ejemplo, una barra de estado que avanza o mensajes del estilo "Tu orden está en preparación" (posible implementación: el cliente abre un canal WebSocket unido a su pedido, y recibe evento cuando cocina lo marca "listo").
7. **Entrega:** Cuando la cocina marca el pedido como listo, el sistema envía una notificación final al cliente: "Tu pedido está listo para recoger." El cliente muestra su número al cajero/cocinero y recoge su comida. El cajero marca el pedido entregado. En el frontend cliente, se puede mostrar un mensaje de cierre "¡Buen provecho! 🍽️". Si el cliente continuaba en el chat, la IA podría despedirse con un mensaje cordial dominicano ("¡Disfruta tu comida! Cualquier cosa, aquí estamos, ¡buen provecho!").

2. Flujo Interno de Preparación y Entrega (Cocina y Caja)

1. **Pedido Entrante en Cocina:** Cuando llega un nuevo pedido, el **Orders Service** del backend emite un evento usando Redis Pub/Sub, que es captado por el **Gateway de WebSocket** de NestJS. Este gateway difunde vía Socket.io el evento `order:new` a todos los clientes conectados con rol cocinero. La interfaz de cocina (abierta en un navegador/terminal en la cocina) tiene un listener para `order:new` y al recibirlo agrega la orden a la lista en pantalla, acompañada de un sonido breve.
2. **Preparación:** El cocinero ve los detalles e inicia la preparación. Puede tocar un botón "En preparación" en la interfaz, lo que podría opcionalmente cambiar el color del pedido (indicativo) y enviar un evento `order:status` con status "preparing". El sistema puede registrar timestamp de inicio. (Esta etapa es más informativa; si se omite, igualmente al marcar listo se computa duración).
3. **Pedido Listo:** Una vez terminada la preparación, el cocinero pulsa "Marcar como Listo". Esto actualiza el estado en la base de datos a "ready". Inmediatamente, el backend emite un evento `order:ready` dirigido al cliente correspondiente (si está conectado, filtrado por ID de pedido) y a los cajeros. El monitor de cocina podría ahora mover este pedido a una sección "Listos para entregar" o simplemente señalarlo como listo.
4. **Notificación al Cliente:** Si el cliente está esperando en una zona con pantalla (o en su propio dispositivo viendo el estado), recibe la notificación de pedido listo. En restaurantes con pagers o pantallas públicas, se podría también reflejar ahí (fuera del alcance actual). En nuestro sistema, asumimos el cliente sabe vía su teléfono o porque el cajero lo llama.
5. **Cobro y Entrega:** El cajero verifica si el pedido ya fue pagado. Si el cliente viene a recoger y no ha pagado, realiza el cobro en ese momento (registrando en sistema). Luego entrega la comida. Finalmente, el cajero marca la orden como "Entregada/Completada" en su interfaz. Esto concluye el ciclo, y la orden ya no aparece en pendientes de cocina ni caja. El administrador la verá en historial con todos sus timestamps (entrada, listo, entregado).

3. Flujo de Administración del Menú

1. **Login Admin:** El administrador inicia sesión en la aplicación (habrá un sistema de autenticación para acceder a las secciones de admin; podría ser formulario de login separado).
2. **Acceso a Menú:** Navega a la sección de "Gestionar Menú". Se le presenta la lista de categorías existentes y platos por categoría.
3. **Agregar/Editar Ítem:** Para añadir un nuevo platillo, pulsa "Nuevo Producto", llena un formulario con nombre, descripción, precio, sube una imagen (opcional), categoría asignada y

disponibilidad. Al guardar, el sistema valida datos (ej. precio numérico, campos obligatorios). El backend (NestJS) recibe la solicitud POST para crear el item. Prisma ORM inserta el registro en PostgreSQL. El servicio de menú emite un evento `menu:updated` con el nuevo item. Las interfaces de cliente (si hay alguna abierta) podrían así actualizarse, pero en muchos casos bastará que futuras aperturas ya lo muestren. (Podemos implementar que clientes conectados escuchen eventos de menú para cambios en vivo).

4. **Desactivar Ítem:** El admin puede, en vez de borrar un platillo, marcarlo "No disponible". Esto cambia una propiedad del item. Nuevamente, se emite `menu:updated` evento. Los frontends de cliente podrían filtrar items no disponibles o mostrarlos atenuados con etiqueta "agotado". La IA asistente también consultará solo items disponibles al responder.
5. **Revisar Dashboard:** El admin navega al dashboard principal para ver métricas. Genera un rango de fechas o ve el día actual. El frontend hace requests GET a endpoints como `/stats/sales?date=today` o suscripciones a eventos agregados, y muestra gráficos. Por ejemplo, un gráfico de líneas comparando ventas de esta semana vs semana pasada, etc. Puede identificar qué platos son más populares (pudiendo usarlo para futuras promociones). Todo esto se logra con consultas predefinidas en el backend (aplicando agregaciones SQL vía Prisma).
6. **Administrar Personal:** El admin visita la sección de usuarios. Crea una cuenta para un nuevo cajero ingresando nombre, email y asignando rol "cajero". El sistema envía un correo de invitación (si configurado) o le da una contraseña temporal. Similar para cocineros. Puede desactivar cuentas (soft-delete o flag "inactive"). Estas operaciones son sencillas CRUD con la tabla de usuarios/roles. NestJS usará mecanismos de seguridad (guards o role decorators) para asegurar solo admin acceda aquí.

Integración de la Inteligencia Artificial (Asistente Virtual)

Una característica central de **Delicia** es el **asistente virtual inteligente** que ayuda a los clientes en el proceso de pedido mediante conversación natural. A nivel técnico, esta IA se implementará integrando un modelo de lenguaje (LLM, como GPT) con nuestro backend a través de **Model Context Protocol (MCP)**.

¿Qué es MCP y por qué usarlo? MCP es un protocolo abierto que permite conectar aplicaciones de lenguaje (LLMs) con fuentes de datos y herramientas externas de forma estandarizada. En esencia, define una forma segura para que nuestra aplicación exponga ciertas funcionalidades (llamadas *tools* o herramientas) que el agente de IA puede invocar durante una conversación. Por ejemplo, crearemos herramientas como: `getMenu(category)`, `findItem(name)`, `createOrder(items)` que la IA usará para consultar el menú en tiempo real o registrar un pedido en la base de datos. Estas herramientas son básicamente funciones de backend que el protocolo MCP hace accesibles al modelo de IA.

Arquitectura del Agente: Utilizaremos el SDK de MCP para TypeScript en el backend, configurando un **servidor MCP** dentro de NestJS. El flujo será: cuando un cliente envía un mensaje en el chat, el backend lo pasa al agente de IA (posiblemente alojado en un servicio externo tipo OpenAI/Anthropic, o local). El agente analiza la intención, y si necesita datos del sistema (por ejemplo, el usuario preguntó "¿Qué tengo en mi pedido hasta ahora?"), el agente formulará una llamada a una de las herramientas MCP. El SDK MCP recibirá esa solicitud, ejecutará la función correspondiente (e.g., obtener items actuales en carrito) y retornará el resultado al agente. El agente entonces responde al usuario incorporando esos datos ("Llevas 1 Sandwich Especial y 1 Jugo de Piña. ¿Deseas algo más?"). Todo esto ocurre en segundos y de forma transparente para el usuario.

Diseño Conversacional: Definiremos *prompts* y reglas para la IA de modo que tenga la "personalidad" adecuada: cordial, eficiente y con un tono local amigable. Por ejemplo, al inicio podría saludar: "*¡Hola!*

Soy tu asistente de pedidos. ¿En qué te ayudo hoy?" y siempre usar lenguaje respetuoso pero cercano, incluyendo dominicanismos suaves si apropiado (ej. "claro que sí, tenemos jugo de chinola riquísimo"). La IA debe manejar: **consultas de menú** (ingredientes, recomendaciones), **toma de pedido** (añadir/quitar items), **consultas generales** (horarios, dirección del local – estas se podrían contestar con datos fijos configurados), y **seguimiento de orden** (estado del pedido, aunque esto último quizás lo maneje más la UI que la IA). Si el usuario formula preguntas fuera del ámbito (por ej., algo no relacionado al restaurante), la IA redirigirá educadamente la conversación al tema de pedidos.

Seguridad y Limitaciones de la IA: Dado que la IA puede ejecutar funciones de nuestro sistema, implementaremos salvaguardas: las herramientas expuestas tendrán verificaciones (por ejemplo, `createOrder` solo puede ejecutarse si el usuario tiene un carrito válido y no se ha creado ya un pedido; no permitiremos a la IA acceder a funciones administrativas). Además, MCP sugiere considerar las herramientas como potencial "arbitrary code execution" y tener precaución. Restringiremos lo que la IA puede hacer – básicamente limitarla a operaciones de lectura de menú y escritura de pedidos dentro de los parámetros esperados. Cualquier respuesta final al cliente pasará por un filtro simple para evitar respuestas inapropiadas. En resumen, la **IA actuará como un agente limitado al dominio de pedidos**, orquestando las funciones disponibles pero sin acceso libre a datos sensibles más allá del menú y órdenes del usuario actual.

Ejemplo Breve (Conversación):
Usuario: *"¿Qué me recomiendas de postre?"*
Asistente IA: *(llama internamente* `getMenu("Postres")` *para listar postres)*
IA: *"Tenemos Bizcocho dominicano tradicional y Helado de guanábana. El bizcocho es esponjoso con suspiros, ¡muy popular! ¿Te apetece uno de estos?"*
Usuario: *"Suena bien el bizcocho. Agrégalo a mi orden."*
Asistente IA: *(invoca* `createOrderItem("Bizcocho dominicano")` *añadiendo al carrito del usuario)*
IA: *"¡Listo! Añadí un Bizcocho dominicano. Tu orden tiene 3 items en total. ¿Deseas algo más o lo confirmamos?"*
...
(Obsérvese cómo la IA usa herramientas para obtener info actualizada y tomar acciones según lo que el usuario dice, manteniendo el contexto de la conversación.)

Arquitectura Técnica y Diseño del Sistema

Visión General: La plataforma se implementará como un **monolito modular**: una única aplicación backend NestJS que contiene múltiples módulos lógicos (cada uno encapsulando controladores, servicios, entidades relacionadas a un subdominio: pedidos, menú, usuarios, etc.). Esta elección facilita el desarrollo inicial y mantiene consistencia, a la vez que prepara el camino para una posible futura transición a microservicios si fuese necesario (cada módulo podría extraerse en un servicio independiente más adelante). La comunicación entre módulos dentro del monolito seguirá principios de bajo acoplamiento – preferentemente a través de interfaces o eventos internos en lugar de llamadas directas fuertemente acopladas, para que el código sea mantenible y testeable.

Stack Tecnológico Resumido:
- **Backend:** NestJS (Node.js/TypeScript) – Framework modular y escalable; PostgreSQL – Base de datos relacional central; Prisma – ORM para modelar y consultar la DB de forma segura; Redis – Base de datos en memoria usada aquí para cachear ciertos datos y principalmente para su sistema Pub/Sub; WebSockets (Socket.io) – canal bi-direccional para features en tiempo real (notificaciones de pedidos, chat quizás).
- **Frontend:** React 18+ con Vite – para una SPA/MPA reactiva; Tailwind CSS – para estilos utilitarios

rápidos y consistentes; ShadCN UI – colección de componentes preconstruidos estilizados con Tailwind (basados en Radix UI) que acelerarán la creación de una UI coherente; TypeScript – en frontend también, para mantener tipados los datos que vienen del backend (ayuda a evitar errores, al igual que en backend).
- **Infraestructura & DevOps (mención breve):** Si bien inicialmente podría correr en un solo servidor, es importante contenedorizar la app (Docker) para fácil despliegue. PostgreSQL y Redis correrán como servicios separados. Con Redis habilitando escalamiento horizontal, podremos correr múltiples instancias del backend NestJS detrás de un balanceador si la demanda crece, sin perder la sincronización en los eventos de WebSocket (gracias al mecanismo de Redis que compartirá eventos entre instancias). Autenticación probablemente con JWT para un frontend sin estado de sesión en servidor, almacenando tokens de forma segura (httpOnly cookies o similar).

Arquitectura Backend (NestJS Monolito Modular)

Adoptaremos una arquitectura estilo **hexagonal (Ports and Adapters)** para estructurar el backend, que calza muy bien con la filosofía de NestJS. En este enfoque:
- La **lógica de negocio** central (casos de uso como *"crear pedido"*, *"marcar pedido listo"*, *"consultar menú"*) reside en servicios de dominio que **no dependen** de detalles de infraestructura (DB, framework web). Esto nos da un núcleo limpio y testeable.
- Definiremos **interfaces (ports)** para operaciones como acceso a datos (por ejemplo, un `OrderRepository` con métodos `save(order)` o `findPendingByKitchen()`) y para integración con otros servicios (p.ej. un `PaymentGateway` si hubiera pagos en línea, o interfaces para notificación push/email).
- Las implementaciones concretas de estas interfaces serán los **adapters**: en nuestro caso, usaremos Prisma como adapter para la base de datos implementando los repositorios, NestJS WebSockets como adapter para notificaciones en tiempo real, etc. NestJS facilita la inyección de dependencias, por lo que podemos inyectar, por ejemplo, `PrismaOrderRepository` en el servicio de pedidos a través de un token de interfaz, manteniendo la posibilidad de sustituirlo (por ejemplo, por un mock en tests).

Organización en Módulos: Cada contexto principal será un módulo de NestJS: - `AuthModule` (autenticación y autorización JWT, estrategia local, etc.), - `UserModule` (usuarios y roles), - `MenuModule` (categorías y productos), - `OrdersModule` (pedidos y lógica de flujo de estado), - `KitchenModule` (podría ser parte de Orders pero podría manejar sus propios gateways de WS relacionados con cocina), - `PaymentModule` (aunque inicialmente manejamos pagos offline, lo creamos con mínimo esqueleto por si luego integraciones), - `ChatbotModule` o `AIModule` (encargado de la integración MCP y coordinación del agente IA), - `NotificationsModule` (si decidimos centralizar eventos y WebSocket gateways).

Los **controladores (Controllers)** expondrán API REST para las funcionalidades CRUD (ej. `POST /orders`, `GET /menu`, `PUT /menu/item/:id`, etc.) y podrían también manejar algunas rutas para vistas si fuera SSR (no es el caso, nuestro frontend es separado). Adicionalmente, tendremos **Gateways de WebSocket** (NestJS permite con `@WebSocketGateway`) para gestionar las conexiones en tiempo real. Posiblemente tengamos un gateway para el canal general de pedidos, y dentro de él rooms o canales por rol o por pedido. Por ejemplo, el `OrdersGateway` envía eventos a la sala `kitchen` o a un namespace `/kitchen` para todo lo que concierne a cocineros, y a salas específicas de cliente (pedido) para notificar al cliente de su pedido particular.

Ejemplo de Código – Endpoint y Servicio (NestJS): A modo ilustrativo, a continuación un pequeño ejemplo simplificado de cómo luciría parte del módulo de pedidos en NestJS, siguiendo buenas

prácticas: definimos un **DTO** para creación de pedido, un **Controlador** que usa un **Servicio** para la lógica, y ese servicio usa Prisma a través de un **PrismaService** compartido (inyección de dependencia):

```
// orders.dto.ts - DTO de entrada para crear pedido
export class CreateOrderDto {
  customerId?: string;
  items: { productId: number; quantity: number; notes?: string }[];
  // ...otros campos como ubicacion, etc.
}

// orders.controller.ts
import { Controller, Post, Body, UseGuards } from '@nestjs/common';
import { OrdersService } from './orders.service';
import { CreateOrderDto } from './orders.dto';
import { JwtAuthGuard } from '../auth/jwt-auth.guard';

@Controller('orders')
export class OrdersController {
  constructor(private readonly ordersService: OrdersService) {}

  @UseGuards(JwtAuthGuard)  // solo usuarios autenticados (ej. cliente
logueado) pueden ordenar
  @Post()
  async createOrder(@Body() createOrderDto: CreateOrderDto) {
    const order = await this.ordersService.createOrder(createOrderDto);
    return order;  // retorna el pedido creado (o al menos su id y status)
  }
}
```

```
// orders.service.ts
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';
import { CreateOrderDto } from './orders.dto';

@Injectable()
export class OrdersService {
  constructor(private prisma: PrismaService) {}

  async createOrder(dto: CreateOrderDto) {
    // Lógica de negocio para crear un pedido
    // Por simplicidad, creamos el pedido y sus items asociados
    const newOrder = await this.prisma.order.create({
      data: {
        customerId: dto.customerId,
        status: 'PENDING',  // status inicial
        items: {
          create: dto.items.map(item => ({
            productId: item.productId,
            quantity: item.quantity,
```

```
            notes: item.notes || '',
          })),
        },
        createdAt: new Date(),
      },
      include: { items: true },  // incluir items en el objeto retornado
    });
    // Emitir evento de nuevo pedido (usando, por ejemplo, un EventEmitter
interno o gateway)
    // this.eventEmitter.emit('order.created', newOrder);
    return newOrder;
  }

  async updateOrderStatus(orderId: number, newStatus: string) {
    const updated = await this.prisma.order.update({
      where: { id: orderId },
      data: { status: newStatus, updatedAt: new Date() },
    });
    // Emitir eventos según nuevo estado si es relevante
    // ...
    return updated;
  }
}
```

En el código anterior, el patrón a resaltar es: el **Controlador** permanece ligero, delegando al servicio; el **Servicio** maneja la interacción con Prisma (nuestro ORM) para crear registros. Se usan DTOs para validar datos de entrada. La lógica de emitir eventos cuando se crea o actualiza un pedido está indicada (podemos usar `EventEmitter2` de NestJS para eventos internos, o invocar métodos de un Gateway de WS para enviar notificaciones a los clientes conectados). Esta separación de responsabilidades facilita pruebas unitarias (podemos testear `OrdersService` aislado simulando `PrismaService`).

Integración de Prisma ORM: Usaremos Prisma como ORM principal. Prisma nos permite definir el esquema de datos en `schema.prisma` con modelos para `User`, `Product`, `Order`, etc., y genera un cliente tipado para consultar la base de datos. Según la receta oficial de NestJS, tendremos un `PrismaService` injectable que extiende de PrismaClient, manejando conexión y potencialmente middleware de logging. Cada módulo que necesite acceso a BD recibe este PrismaService. Esto centraliza la configuración de la base de datos y evita abrir múltiples conexiones. La **ventaja de Prisma** es la seguridad de tipos y evitar inyecciones SQL, así como facilidad para migraciones. Adicionalmente, podremos aprovechar las transacciones de Prisma (por ejemplo, al crear un pedido con sus items, como es arriba, todo se hace en una llamada y se garantiza atomicidad).

Comunicación en Tiempo Real (WebSockets + Redis): Como se mencionó, utilizaremos **Socket.io** en NestJS para la capa de WebSocket. Socket.io por defecto permite emitir eventos a todos los clientes o a ciertos canales. En nuestro diseño, al iniciar NestJS, configuraremos un **adapter** de Socket.io que use Redis (NestJS tiene documentación para integrar `socket.io-redis` adapter). Esto es crucial para soportar múltiples instancias de servidor: una instancia publica un evento de nuevo pedido, Redis lo recibe y lo publica a las demás instancias, así *todos* los websockets conectados reciben el evento. Desde el punto de vista del código, una vez configurado, podemos hacer

`this.server.emit('order:new', payload)` en el gateway, y el adapter se encarga de la distribución clusterizada.

- *Ejemplo:* En el `OrdersService.createOrder` mostrado, después de crear el pedido podemos inyectar el gateway (o usar EventEmitter) para notificar:

```
this.ordersGateway.broadcastNewOrder(newOrder);
```

Y en `OrdersGateway` implementar:

```
@WebSocketGateway({ namespace: '/kitchen' })
export class OrdersGateway {
  @WebSocketServer() server: Server;
  broadcastNewOrder(order) {
    this.server.emit('order:new', order);
  }
  // ... también métodos para broadcast de status updates, etc.
}
```

Con `namespace: '/kitchen'` nos aseguramos que solo los clientes conectados al namespace de cocina reciban ese evento (los cocineros). Para clientes individuales, podríamos usar rooms nombradas con el id de pedido o id de usuario.

Autenticación y Autorización: NestJS facilitará la implementación de JWT Auth. Tendremos un módulo Auth con estrategia local (login con usuario/contraseña) que al validar emite un JWT firmado. Los clientes almacenan ese token (p. ej., en una cookie segura) y lo incluyen en cada petición subsiguiente. Para websockets, Socket.io puede enviar el token en el handshake query o headers para también autenticar la conexión. Utilizaremos **Guards de NestJS** para proteger rutas según rol – por ejemplo, el controlador de admin solo accesible con rol admin, etc. Los roles se pueden manejar con un decorador custom `@Roles('admin')` más un guard que chequea `req.user.role`. De esta manera, se asegura que cada rol solo acceda a sus pantallas (también controlado en frontend con rutas protegidas). Contraseñas de usuarios se almacenarán **hasheadas** (usando bcrypt).

Manejo de Errores: El backend contará con filtros de excepción globales para retornar errores formateados (Nest trae uno por defecto para HttpException). Por ejemplo, si un usuario no autorizado accede a recurso prohibido, devuelve 403 JSON apropiado; validaciones fallidas de DTO devuelven 400 con mensajes; etc. Esto ayudará al frontend a reaccionar (mostrar mensajes amigables al usuario final).

Arquitectura Frontend (React + Vite)

El frontend será una **SPA (Single Page Application)** construida en React, lo que brinda una experiencia fluida al usuario sin recargas completas. Gracias a Vite, tendremos tiempos de carga y desarrollo muy rápidos. Usaremos **componentes funcionales con Hooks** para el manejo de estado y efectos secundarios. La estructura del proyecto podría organizarse por *features* o *pages*. Ejemplo:
- Carpeta `src/pages` con componentes de página: `MenuPage.tsx`, `CheckoutPage.tsx`, `KitchenDashboard.tsx`, `AdminDashboard.tsx`, etc.
- Carpeta `src/components` para componentes reutilizables: `MenuItemCard.tsx`, `OrderList.tsx`, `ChatbotWidget.tsx`, `DashboardChart.tsx`, etc.
- Carpeta `src/context` si usamos Context API para ciertos estados globales (ej. contexto de

autenticación para conocer usuario logueado y rol en toda la app; contexto de carrito de compras para persistir la orden en proceso).
- Podríamos utilizar una librería de estado global como Zustand o Redux si la complejidad crece, pero inicialmente tal vez context + hooks basten.

Estilo y Componentes (Tailwind + ShadCN UI): Adoptaremos Tailwind CSS para desarrollar un diseño responsive y consistente rápidamente. Tailwind nos permite aplicar clases utilitarias (p.ej. `bg-indigo-600 text-white p-4 rounded`) directamente en los elementos JSX para estilarlos. Sin embargo, para evitar exceso de clases en elementos complejos, utilizaremos componentes preconstruidos de **ShadCN UI** – que básicamente provee componentes de interfaz (botones, diálogos, menús desplegables, pestañas, tablas, etc.) ya estilizados con Tailwind siguiendo un diseño moderno y accesible. Estos componentes se pueden personalizar para que encajen con nuestra identidad visual. Por ejemplo, ShadCN nos da un componente `<Button>` ya bonito y accesible, que podemos extender con nuestras variantes de color (quizá un verde o naranja particular representativo de la marca dominicana que creemos).

Se definirá un **tema de colores** Tailwind en su configuración, posiblemente inspirado en la bandera dominicana (azul añil, rojo vibrante, blanco) pero adaptado a UI (tonos un poco más suaves para no ser estridentes en pantalla). También incluiremos colores tropicales complementarios (verdes, amarillos tipo sol/plátano) para acentos en gráficas o alertas. La **tipografía** será clara y moderna, priorizando legibilidad; se puede usar una fuente sans-serif geométrica o incluso alguna que tenga buena presencia en español. Importante: internacionalización por ahora no es prioridad (asumimos la app en español dominicano), pero al usar Unicode y fuentes adecuadas, caracteres especiales estarán bien.

Responsive Design: La aplicación debe funcionar tanto en un móvil/tablet (quizá para pequeños puestos que usan una tablet), como en una pantalla grande (admin en computadora, o pantalla en cocina). Utilizaremos unidades fluidas y breakpoints de Tailwind para adaptar layouts. Por ejemplo, la barra lateral de admin quizá se muestre contraída en móvil, etc. El *Modo Cocina* probablemente esté optimizado para una pantalla horizontal grande, pero haremos que al menos sea scrollable en dispositivos más pequeños si se diera el caso.

Interacción con Backend API: Para obtener o enviar datos (cuando no se usan websockets), utilizaremos la API REST. Podemos emplear `fetch` nativo o una librería como Axios. Gracias a TypeScript, definiremos tipos o interfaces para las respuestas (p. ej. `Order` interface matching the backend DTO) para tener auto completado y chequeos en compile time. Las llamadas se harían típicamente dentro de hooks (e.g., useEffect para fetch on load) or via event handlers.

Manejo de WebSocket en Frontend: Usaremos la librería cliente de Socket.io para conectarnos al servidor WebSocket. Al montar ciertas componentes, estableceremos la conexión y escucharemos eventos. Por ejemplo, en `KitchenDashboard.tsx` (pseudocódigo):

```
import { useEffect, useState } from 'react';
import { io, Socket } from 'socket.io-client';

function KitchenDashboard() {
  const [orders, setOrders] = useState<Order[]>([]);
  useEffect(() => {
    const socket: Socket = io('<SERVER_URL>/kitchen', {
      auth: { token: localStorage.getItem('token') }  // enviar token JWT
para autenticar
```

```
    });
    socket.on('order:new', (order: Order) => {
      setOrders(prev => [ ...prev, order ]); // agregar nuevo pedido a la
lista
    });
    socket.on('order:update', (update: {id: number, status: string}) => {
      setOrders(prev => prev.map(o => o.id === update.id ? {...o, status:
update.status} : o));
    });
    return () => { socket.disconnect(); };
  }, []);

  // renderizar la lista de orders...
}
```

En el fragmento anterior, establecemos conexión al namespace `/kitchen` y escuchamos dos eventos: nuevos pedidos y actualizaciones de estado. Al recibirlos, actualizamos el estado local `orders` para re-renderizar la UI con los cambios. Notar que enviamos el JWT en la conexión (`auth` option) para que el servidor valide que solo usuarios con rol cocinero entren al canal de cocina.

Ejemplo de Componente React (UI de Item de Menú): Para ilustrar el uso de Tailwind y ShadCN, imagine un componente para mostrar un platillo en la lista:

```
// MenuItemCard.tsx
import { Button } from "@/components/ui/button"  // importado de ShadCN UI
import { Badge } from "@/components/ui/badge"

function MenuItemCard({ item, onAdd }) {
  return (
    <div className="bg-white shadow rounded-md p-4 flex flex-col">
      <h3 className="text-xl font-semibold mb-1">{item.name}</h3>
      <p className="text-sm text-gray-600 flex-grow">{item.description}</p>
      <div className="mt-2 flex items-center justify-between">
        <span className="text-lg font-bold">${item.price}</span>
        {item.isAvailable ? (
          <Button onClick={() => onAdd(item)} className="ml-2">Añadir</
Button>
        ) : (
          <Badge variant="destructive">Agotado</Badge>
        )}
      </div>
    </div>
  );
}
```

En este componente, usamos clases Tailwind (`bg-white`, `shadow`, `rounded-md`, etc.) para la tarjeta. Usamos un `<Button>` de ShadCN UI para el botón Añadir (que ya viene estilizado y accesible) y un `<Badge>` para mostrar si está agotado. Esto demuestra cómo mantenemos el código declarativo y limpio, delegando el estilo a utilidades y componentes pre-hechos.

Microinteracciones y Feedback Visual: Aprovecharemos CSS transitions y pequeñas animaciones para dar fluidez. Ejemplos: cuando el usuario añade un item, podríamos hacer que el botón "Añadir" muestre un check ✓ por un segundo y vuelva a estado normal, indicando que se añadió. Al abrir el chat de la IA, podría deslizarse desde un lado con una animación. Al marcar un pedido como listo en cocina, ese pedido podría parpadear en verde suave. Estas microinteracciones aumentan la percepción de calidad. Usaremos las capacidades de Tailwind para transitions (`transition-colors`, `duration-300`, etc.) y keyframes CSS personalizados donde necesario.

Identidad Gráfica Dominicana: Buscamos que la aplicación se sienta **moderna y local**. Esto podría lograrse incorporando sutilmente algunos elementos: por ejemplo, íconos personalizados inspirados en comida típica (un icono de empanada, un pote de habichuelas, etc.) para adornar categorías o el splash screen. Colores vibrantes pero equilibrados (azul cerúleo como el mar de Punta Cana, verde hoja de palma, naranja atardecer del Malecón) pueden aparecer en botones o encabezados. Cuidaremos que no distraigan de la usabilidad. También el tono comunicativo en textos y mensajes será cercano: usando expresiones coloquiales ligeras (de acuerdo al público objetivo dominicano) para generar empatía. Todo esto sin caer en estereotipos, más bien resaltando la calidez y alegría asociada a la cultura dominicana.

Consideraciones de Escalabilidad y Mantenibilidad

- **Modularidad para Escala Futura:** Aunque es un monolito, la separación en módulos lógicos facilita asignar equipos distintos a diferentes áreas sin pisarse, y posibilita escalar componentes específicos. Por ejemplo, si el volumen de pedidos crece enormemente, podríamos escalar horizontalmente el módulo de pedidos (replicando instancias de la app, ya que es stateless salvo websockets, los cuales manejamos con Redis). Si en un futuro se decide extraer, por ejemplo, el Chatbot como microservicio independiente, la arquitectura de puertos y adaptadores ayuda porque la lógica de IA está aislada en `AIModule` con interfaces claras hacia el resto.
- **Redis Cache:** Además de Pub/Sub, podemos usar Redis para cachear datos frecuentemente leídos pero poco cambiantes, como el menú o las estadísticas del día, reduciendo carga a PostgreSQL. Esto será útil en horas pico. Implementaremos caching en capas de servicio donde tenga sentido (p. ej., cache de 1 minuto para el listado de menú si muchos usuarios lo piden simultáneo).
- **Clean Code y Contribuciones:** Fomentaremos buenas prácticas de código: linters (ESLint), formateo (Prettier), y seguimiento de convenciones de NestJS (archivos bien nombrados, métodos cortos y claros). También, cubriremos partes críticas con **pruebas unitarias** (especialmente la lógica de pedidos y las herramientas de IA) y pruebas de integración básicas (simular un flujo completo de pedido). Esto garantiza que al agregar nuevas funcionalidades (como las promociones en el futuro) no rompamos lo existente.
- **Seguridad:** Aplicaremos medidas de seguridad como sanitización de inputs (Nest ya ayuda con validation pipes contra data mal formada). Protegemos datos sensibles: por ejemplo, si se guardan datos de tarjetas (no contemplado ahora), se haría de forma segura o delegada a terceros. Evitaremos exponer información de más en las APIs – por ejemplo, un cliente autenticado solo puede ver sus propios pedidos, el servidor validará el `userId` en cada consulta de pedidos. También implementaremos *rate limiting* en endpoints públicos si fuera necesario (para evitar abusos del chat por bots, etc.).
- **Logs y Monitoreo:** Instrumentaremos el backend para logging de eventos importantes (un pedido creado, error de DB, etc.) y potencialmente uso de un APM or monitoring (como Grafana/Prometheus) if deployed in production. Esto ayuda a mantener la plataforma confiable y diagnosticar problemas rápido.

Plan de Implementación y Tareas por Equipo

Para llevar a cabo el desarrollo de **Delicia**, dividiremos el trabajo entre los equipos **Backend** y **Frontend**, manteniendo sincronía en los contratos (API endpoints, formatos de datos, eventos) y con iteraciones para integrar y probar funcionalidades incrementalmente. A continuación se listan las tareas principales de cada área:

Equipo Backend (NestJS & Base de Datos)

- **Configuración Inicial del Proyecto:** Crear la base del proyecto NestJS (usando Nest CLI), configurar el **monorepo modular** con TypeScript. Instalar dependencias clave: `@nestjs/websockets`, `socket.io`, `@nestjs/config` (para variables de entorno), `@nestjs/jwt` (auth), `prisma`. Inicializar Prisma with a PostgreSQL connection.
- **Diseño del Esquema DB (Prisma):** Definir el modelo de datos en `schema.prisma` incluyendo tablas: Users (campos: id, nombre, email, rol, password hash, etc.), Categories, Products, Orders, OrderItems, etc. Ejecutar migraciones para crear las tablas en la base de datos PostgreSQL.
- **Implementar Módulo de Autenticación:** Configurar JWT AuthStrategy. Crear endpoints `POST /auth/login` (devuelve token) y `POST /auth/signup` (posiblemente para registro de clientes, aunque quizás se maneje sin registro). Proteger rutas posteriores con `JwtAuthGuard`. Verificar que los roles se incluyan en el token JWT para usar en autorización.
- **Implementar Módulo de Usuarios/Roles:** CRUD de usuarios básico (principalmente para que admin cree personal). Seeds iniciales: crear un usuario admin por defecto. Endpoint `GET /users/me` para que frontend obtenga perfil y rol. Guards para restringir endpoints admin.
- **Implementar Módulo de Menú:** Endpoints REST para categorías (`GET /menu/categories`, `POST /menu/categories`, etc.) y productos (`GET /menu/items`, `POST /menu/items`, `PUT /menu/items/:id`, `DELETE /menu/items/:id`). Servicios Prisma para leer/escribir. Considerar añadir filtrar solo disponibles en `GET /menu/items` para cliente vs admin que ve todos. Incluir validación de DTOs (usando `class-validator`).
- **Implementar Módulo de Pedidos:** Endpoints `POST /orders` (crear pedido), `GET /orders/:id` (detalle, con items y estado), quizás `GET /orders` (lista filtrada por usuario o por estado para admins). Lógica de negocio: al crear pedido, asignar código/numero; si soportamos multi-sucursal en futuro, incluirla. Emitir evento de nuevo pedido (integrar con WebSocket gateway). Endpoint para actualizar estado (`PUT /orders/:id/status`) que podrían llamar cocina o caja (con autorización por rol).
- **Configurar WebSocket Gateway:** Crear clase OrdersGateway con namespace o filters por rol. Integrar `IoAdapter` de socket.io con Redis. Testear que cuando un pedido se crea, el evento llega a un cliente simulado suscrito. Similar para update de estado.
- **Integración IA (MCP):** Montar el servidor MCP. Definir las herramientas: por ejemplo, dentro de `AIModule`, usar el SDK TS de MCP para registrar herramientas `getMenu`, `createOrder`, etc. Probablemente necesitemos definir cómo el agente IA se comunica con nosotros; podría ser que actuamos como tanto cliente como servidor MCP. Investigar mejor implementación: quizá ejecutar un cliente LLM (OpenAI API) que llame a nuestras tools definidas via MCP. Implementar un servicio que orquesta esto: recibe prompt del usuario, llama al agente (via SDK), espera respuesta enriquecida. *Tarea específica:* escribir prompts iniciales y funciones en JSON schema para herramientas, y probar con un modelo (puede ser offline primero). Asegurar la seguridad (herramientas no hacen más de lo debido).
- **Servicios de Notificación (opcional):** Si tiempo permite, integrar un EventEmitter interno para manejar eventos de dominio (like `order.created`) de forma desacoplada: por ejemplo, al ocurrir, un listener envía notificación email (futuro) o logs. Esto es extra; enfoque principal son websockets.

- **Pruebas Unitarias y de Integración (Backend):** Escribir tests para los servicios principales (OrdersService, MenuService, etc.), usando either an in-memory SQLite DB for Prisma or test transactions. Pruebas de autorización (que endpoints protegidos rechacen usuarios sin rol). Prueba simulada del chat IA (dado cierta entrada, se espera llame a tool X).
- **Documentación API:** Crear documentación de la API (quizá integrar Swagger en NestJS para listar endpoints, dto schema, etc. para uso interno y para que frontend consulte si duda formato).

Equipo Frontend (React & UI/UX)

- **Setup del Proyecto React:** Inicializar proyecto con Vite + React + TypeScript. Configurar TailwindCSS (postcss, etc.) y ShadCN UI (instalar los componentes necesarios). Organizar estructura de carpetas base. Configurar router (React Router) con rutas protegidas (por rol). E.g., public routes: login, ordering interface; private routes: /kitchen, /admin, /cashier (que requieren auth & specific role).
- **Implementación de Autenticación UI:** Crear páginas de Login y Registro (si aplica) con formularios usando componentes de formulario de ShadCN (inputs, etc.). Conectar al endpoint `/auth/login` para obtener JWT; guardar token (posiblemente en localStorage o cookie). Gestionar estado global de auth (context or simple state in App). Redirigir usuarios a la página adecuada según rol tras login (ej. admin->/admin, cocinero->/kitchen, cajero->/cashier, cliente->/menu).
- **Pantalla de Menú y Pedido (Cliente):** Desarrollar la página principal de pedido del cliente. Esto incluye: mostrar categorías y lista de productos. Implementar el componente `MenuItemCard` para cada plato (como el ejemplo de código). Añadir funcionalidad al botón “Añadir”: actualiza estado del carrito global (puede usar Context `CartContext`). Mostrar un resumen del carrito en un sidebar o modal, con botón “Confirmar Pedido”. Al confirmar, llamar al API `/orders` con los datos; manejar respuesta (navegar a página de confirmación/seguimiento).
- **Chatbot UI (Cliente):** Implementar un widget de chat flotante o embebido. Por simplicidad, podría ser un componente que aparece sobre la UI de menú. Este componente `ChatbotWidget` tendrá una ventana de chat (lista de mensajes) y un input para texto. Necesita manejar estado de la conversación (lista de {sender, message}). Al enviar un mensaje, llamar a un endpoint del backend `/ai/chat` (por ejemplo) que gestione la interacción con la IA (este endpoint en backend usaría el agente MCP para obtener respuesta). Alternativamente, se podría mantener la conexión por WebSocket para chat, pero quizá REST polling es más simple al inicio. Decidir e implementar la comunicación: si via REST, cada mensaje produce una request y la respuesta se agrega al chat; si via WS, abrir una conexión en este componente al namespace chat, enviar evento y escuchar respuesta. En cualquier caso, formatear la respuesta (puede incluir texto, opciones sugeridas). Asegurar scroll al último mensaje, etc. Añadir pequeñas animaciones: por ej., mientras espera respuesta, mostrar "Escribiendo..." con tres puntitos animados.
- **Vista de Confirmación y Tracking:** Tras confirmar pedido, mostrar número de pedido e instrucciones (“Acércate a caja para pagar” o similar). Si se decide implementar tracking en cliente, esta página se suscribe vía WebSocket a eventos `order:update` filtrados a su pedido. El backend podría emitírselos uniéndolo a una sala con su pedidoId cuando hizo el pedido (necesitaríamos en OrdersGateway, cuando un cliente crea pedido, añadir su socket a room `order_<id>`). Simplificadamente, se puede hacer polling en frontend cada X segundos al endpoint `/orders/:id` para ver si status cambió. Pero WebSocket es más en tiempo real; implementar el escucha si es viable.
- **Interfaz de Cocina (KitchenDashboard.tsx):** Crear la página para cocineros. Inicialmente cargar la lista de pedidos pendientes (GET `/orders?status=PENDING` por ejemplo). Mostrar en un componente `OrderCard` información condensada. Implementar la lógica de conexión

WebSocket (como ejemplificado antes) para recibir nuevos pedidos y actualizaciones. Añadir controles: un botón en cada OrderCard "Marcar Listo" (que hará PUT `/orders/:id/status` a "READY"; podemos optimísticamente actualizar UI de inmediato y el WS después lo confirmará). Considerar confirmación modal para evitar clics accidentales. UI/UX: usar colores de estado (ej. PENDING = amarillo, READY = verde).

- **Interfaz de Caja (CashierDashboard.tsx):** Similar a cocina: lista de pedidos que requieren atención del cajero. Quizá filtrar por `status != COMPLETED`. Actualizar vía WS también. Mostrar claramente cuales necesitan cobro (status PENDING && not paid flag) vs cuales están listos para entregar (READY status). Para cobro: un botón "Marcar Pagado" en pedidos no pagados (realiza PUT `/orders/:id/pay` o podríamos fold payment into status "PAID" if combine). Para entrega: botón "Entregado" (PUT `/orders/:id/status` -> COMPLETED). Podrían estar en secciones separadas "Por cobrar" y "Para entregar". Tras marcar completado, remover de lista.
- **Interfaz de Admin (AdminPanel):** Implementar varias sub-páginas o secciones:
- **Dashboard:** con tarjetas resúmen (total ventas hoy, pedidos hoy, etc.) y gráficas. Empezar con una o dos gráficas simples (ej. usando Chart.js React wrapper). Datos los obtenemos de endpoints `/stats` que el backend deberá proveer. Si esos no están listos, se pueden mockear temporalmente.
- **Gestión de Menú:** una tabla o lista de productos editables. Usar componentes de tabla de ShadCN UI si hay (o simple HTML table styled). Cada fila con acciones editar/eliminar. Un formulario para nuevo producto (puede ser modal emergente o página aparte). Para cargar datos: GET `/menu/items` (admin vería todos). Para editar: abrir modal con formulario, al guardar hacer PUT. Manejar la respuesta actualizando la lista local sin requerir refetch completo (o simplemente refetch tras operación, dado pocos datos no es costoso).
- **Gestión de Usuarios:** listar empleados, con botón "Agregar usuario" (form para email, rol, etc.). Llamar API correspondiente. Permitir desactivar usuarios con toggle.

- **Configuraciones:** uno o dos campos editables (horario, nombre restaurante) para demostrar capacidad, aunque su uso en otras partes sea limitado al inicio.

- **Estilos y Pulido:** Ajustar CSS para que todas las páginas tengan un diseño cohesivo. Testear en distintos tamaños de pantalla. Añadir logos o íconos temporales (un logo "Delicia" temporal con un emoji de comida quizás) solo para demostrar la marca. Implementar los textos finales, asegurándose que el tono de comunicación esté presente (por ejemplo mensajes de éxito: "¡Listo! Los cambios se han guardado." en vez de solo "Success"). Revisar accesibilidad: colores contrastantes, alt en imágenes.

- **Pruebas en Frontend:** Realizar pruebas manuales exhaustivas en el flujo completo (pedido -> cocina -> caja -> finalizado). Si es posible, escribir algunos unit tests para componentes puros (por ejemplo, lógica de añadir items al carrito, se espera que sume cantidades correctamente). También probar la integración con backend simulado (usando un servidor de prueba o mocks) para asegurarse de manejo de estados de conexión (ej. WS desconectado muestra alerta, etc.).
- **Iteración de Feedback:** Hacer una demo interna de la aplicación, recopilar feedback de usabilidad (por ejemplo, ¿entiende el usuario promedio que puede usar el chat para pedir? ¿deberíamos resaltar ese feature con un tooltip?). Incorporar mejoras UI/UX según el tiempo restante.

Finalmente, ambos equipos coordinarán para **integración final**: conectar el frontend con el backend desplegado en entorno de pruebas, y garantizar que todas las piezas (pedidos, IA, realtime, etc.) funcionan en conjunto de forma fluida. Cada tarea estará asociada a entregables tangibles (por ejemplo, *"Endpoint X funcionando"*, *"Componente Y implementado"*, *"chatbot responde recomendaciones"*), lo

que permitirá seguir el progreso y ajustar prioridades si fuese necesario. Con esta hoja de ruta, **Delicia** estará encaminada a ser una plataforma innovadora que combine tecnología de punta (IA, tiempo real) con un entendimiento cultural local, proporcionando una solución de pedidos eficiente y encantadora para República Dominicana.

Fuentes: Las decisiones técnicas se fundamentaron en buenas prácticas y referencias de la industria, por ejemplo la separación por capas y uso de arquitectura hexagonal para mantener bajo acoplamiento, el uso de Prisma ORM por su seguridad de tipos y soporte robusto a PostgreSQL, la adopción de Redis Pub/Sub para garantizar la entrega de eventos en tiempo real en entornos escalables, y la integración de MCP (Model Context Protocol) para dotar al agente de IA de capacidades conectadas al contexto de la aplicación. Estas referencias respaldan la viabilidad técnica del enfoque propuesto y servirán de guía durante la implementación detallada.
