Neobank – Guía de Onboarding y Arquitectura

Onboarding: Introducción al Proyecto

Neobank es una plataforma de banca digital implementada en Go, diseñada para gestionar cuentas financieras de clientes y sus transacciones de forma segura y escalable. El sistema está dividido en múltiples microservicios especializados, lo que facilita su mantenimiento y escalabilidad. A grandes rasgos, Neobank permite registrar clientes, verificar su identidad (KYC), abrir cuentas contables para ellos y procesar transferencias de dinero peer-to-peer (P2P) entre usuarios.

Propósito del proyecto: brindar una solución básica de banca digital que sea modular y extensible. A continuación, se resumen los objetivos generales del sistema:

- **Experiencia de onboarding simple:** Facilitar el registro de nuevos clientes, incluyendo la recopilación de datos y documentos para cumplir con políticas KYC, de forma ágil y comprensible.

- **Cumplimiento y seguridad:** Asegurar la verificación de identidad (Know Your Customer) y validaciones reglamentarias antes de habilitar funcionalidades financieras, protegiendo al mismo tiempo la confidencialidad de los datos.

- **Gestión confiable de cuentas y balances:** Crear y administrar cuentas contables para cada cliente, llevando un libro mayor (ledger) confiable que garantice la integridad de los saldos y transacciones.

- **Transferencias P2P eficientes:** Permitir a los usuarios enviar y recibir fondos al instante entre sí, con garantías de consistencia (cada transferencia se registra con doble partida contable) e idempotencia (evitar duplicados).

- **Arquitectura escalable y modular:** Emplear microservicios desacoplados que puedan evolucionar independientemente, comunicándose mediante APIs eficientes (gRPC) y eventos, lo cual sienta las bases para escalar el sistema y añadir nuevas funcionalidades fácilmente en el futuro.

En resumen, Neobank busca que cualquier desarrollador o equipo pueda entender rápidamente el dominio bancario que cubre, a la vez que provee una base técnica sólida para construir aplicaciones financieras confiables.

Flujo Funcional General del Sistema

El sistema abarca varios flujos funcionales clave que involucran a los diferentes servicios. A continuación, se describe de forma simplificada el recorrido típico desde que un cliente se registra hasta que realiza una transferencia P2P:

1. **Onboarding de clientes:** Un nuevo usuario se registra proporcionando sus datos personales básicos y, de ser necesario, documentación de identidad. El sistema

(principalmente el **account-service**) almacena estos datos y marca al usuario como pendiente de verificación. Este paso inicial crea una entidad de cliente en la base de datos de usuarios (con estado "registrado, pendiente KYC").

2. **Validaciones KYC:** Luego del registro, se activa el proceso KYC (Know Your Customer, "Conoce a tu Cliente"), donde se verifica la identidad y la información del usuario. El **account-service** coordina esta validación, que puede involucrar revisar documentos de identidad, compararlos contra listas de personas bloqueadas, y otros controles reglamentarios. Hasta no completar el KYC, el cliente no puede usar funcionalidades financieras. Una vez que el usuario aprueba las validaciones KYC, su estado cambia a verificado y ya está habilitado para operar.

3. **Creación de cuentas contables:** Al confirmar que el cliente superó KYC, el sistema procede a crear una cuenta contable (ledger account) asociada a ese cliente. El **ledger-service** se encarga de esto: se le solicita (vía gRPC) generar una nueva cuenta en el ledger, típicamente especificando la moneda base para el usuario. El **ledger-service** crea un registro en su base de datos de cuentas contables y la deja activa para su uso. Cada cuenta contable tiene un identificador único y se inicia con saldo cero. En adelante, esta cuenta contable representará los fondos del usuario en la moneda correspondiente. (Nota: un cliente podría tener múltiples cuentas contables si maneja varias monedas o productos, pero cada una se crea por separado según se requiera).

4. **Flujo de transferencias P2P:** Con el cliente ya verificado y con una cuenta contable activa, puede enviar y recibir transferencias. Cuando un usuario inicia una transferencia P2P (por ejemplo, enviar dinero a otro cliente dentro de la plataforma), el **payments-service** toma la solicitud y realiza varias acciones: (a) valida que tanto el remitente como el destinatario existan y estén verificados, (b) consulta el saldo de la cuenta de origen para confirmar que haya fondos suficientes, (c) genera un identificador único de la transacción (usado como externalID de idempotencia) y llama al **ledger-service** vía gRPC para registrar la transferencia. El **ledger-service** al recibir la orden de transferencia, ejecuta una operación de doble entrada: crea dos asientos contables, un cargo (débito) en la cuenta del remitente y un abono (crédito) en la del beneficiario por el mismo monto. Ambos asientos se guardan dentro de una misma transacción de base de datos para garantizar atomicidad. Al final, el **ledger-service** verifica que la suma de los débitos y créditos del movimiento sea cero (balance neto) antes de confirmar, asegurando la integridad contable. Si todo es correcto, retorna el éxito de la operación. El **payments-service** entonces notifica al usuario origen y destino del resultado (por ejemplo, mostrando que la transferencia fue completada). En caso de cualquier fallo (usuario sin fondos, cuentas inexistentes, etc.), la operación se aborta y se informa del error sin que ningún cargo se efectúe.

Diagrama resumido de flujo: Usuario se registra → (KYC) → Usuario verificado → Se crea cuenta contable

* Usuario inicia transferencia → Validaciones → Se registra la transferencia en ledger (doble entrada) → Actualización de saldos → Notificaciones. Cada paso involucra uno o varios

microservicios colaborando, como se detalla en la siguiente sección.

Arquitectura Basada en Microservicios

El sistema está construido bajo una arquitectura de microservicios, donde cada componente se enfoca en un dominio específico de la lógica de negocio. Esto implica que diferentes aspectos (clientes, cuentas contables, pagos, etc.) son manejados por servicios independientes que se comunican entre sí cuando es necesario. A continuación, se presentan los principales componentes y sus responsabilidades, así como la forma en que interactúan:

Account-Service (Servicio de Cuentas y Clientes)

El **account-service** es el microservicio encargado de la gestión de clientes y sus cuentas a nivel de negocio. Sus responsabilidades incluyen: el registro de nuevos usuarios, el manejo del flujo de onboarding, y la coordinación del proceso de KYC. Este servicio mantiene la información del cliente (perfil, documentos, estado de verificación) en su propia base de datos. Una vez que un cliente es verificado, el **account-service** se encarga de solicitar la creación de la cuenta contable correspondiente para el cliente – para ello invoca al ledger-service mediante una llamada gRPC CreateAccount.

Además, es probable que el **account-service** exponga APIs hacia el exterior (por ejemplo, endpoints HTTP/REST) que permitan a aplicaciones cliente (móvil o web) interactuar con el sistema: crear usuarios, consultar su estado, iniciar transferencias, etc. De esta manera, actúa en parte como fachada o puerta de entrada al ecosistema de microservicios, orquestando las llamadas internas necesarias. (En una implementación robusta, a veces se podría utilizar un API Gateway dedicado, pero para simplificar asumimos que el **account-service** cumple ese rol de exposición de servicios al cliente final).

Ledger-Service (Servicio de Libro Mayor)

El **ledger-service** maneja la lógica contable central del neobank. Representa el libro mayor donde se registran todas las cuentas financieras y movimientos (transacciones) de forma consistente. Entre sus funciones específicas están: crear cuentas contables (cuando lo solicita **account-service** u otro componente), registrar transferencias aplicando la doble partida (cargos y abonos) y mantener los saldos actualizados, y proporcionar consultas de balance de una cuenta determinada. Este servicio encapsula las reglas contables fundamentales, por ejemplo: no duplicar transfers (usa un ExternalID para evitar registrar dos veces la misma operación) y garantizar que cada transferencia esté balanceada a cero. El **ledger-service** persiste sus datos en una base de datos PostgreSQL propia (esquema ledger), donde lleva tablas de cuentas, transferencias y asientos contables. Funciona de manera autónoma: expone una interfaz gRPC (servidor corriendo típicamente en el puerto 50051) para que otros servicios le envíen solicitudes. Al recibir una orden (por ejemplo, PostTransfer), ejecuta la lógica en una transacción de base de datos y responde con el resultado. Debido a su diseño independiente, ningún otro servicio accede directamente a la base de datos del ledger; cualquier operación contable debe pasar por las APIs del **ledger-service**, manteniendo así la integridad del sistema contable.

Payments-Service (Servicio de Pagos P2P)

El **payments-service** es el componente dedicado a las transferencias de dinero entre cuentas de usuarios (pagos P2P dentro de la plataforma). Actúa como orquestador de las operaciones de pago: recibe las solicitudes de transferencia (normalmente iniciadas vía **account-service** o directamente desde un cliente), valida las condiciones de negocio y coordina con otros servicios para llevarla a cabo. En concreto, cuando se pide una transferencia, **payments-service** verifica que la cuenta origen tenga saldo disponible (consultando al **ledger-service** el balance, mediante GetBalanceCents por gRPC), comprueba que la cuenta destino exista, y luego invoca al **ledger-service** con la operación PostTransfer para realizar el cargo/abono correspondiente en el libro mayor. Al ser el **ledger-service** quien efectúa realmente el registro contable, **payments-service** se concentra en las reglas de negocio previas y posteriores, como controlar límites de monto, gestionar notificaciones de pago enviado/ recibido, y manejar posibles reintentos o reversos en caso de errores. Este servicio también podría generar eventos tras una transferencia exitosa (por ejemplo, un evento "TransferCompleted") para que otros servicios se enteren de la transacción realizada. Nota: en algunas arquitecturas, el **payments-service** podría ser opcional (las transferencias P2P podrían manejarse solo con **account-service** + **ledger-service**); sin embargo, separar un servicio de pagos permite aislar la lógica específica de transferencias (p. ej. futuros pagos a terceros, integraciones con otros bancos, etc.) y mantener el principio de responsabilidad única.

Comunicación e Integración entre Servicios

Los microservicios de Neobank se comunican entre sí de dos formas principales:

- **Llamadas gRPC (sincrónicas):** Para operaciones directas y consultas inmediatas, se utilizan Remote Procedure Calls sobre gRPC. Por ejemplo, el **account-service** llama al método remoto CreateAccount del **ledger-service** para crear una nueva cuenta contable al instante, o el **payments-service** invoca PostTransfer en **ledger-service** para ejecutar una transferencia. Estas llamadas son sincrónicas: el servicio emisor espera la respuesta del receptor para continuar, lo que es adecuado cuando se necesita confirmar una acción en tiempo real (e.g., informar al usuario que su transferencia fue realizada con éxito). Los mensajes y datos se serializan usando Protocol Buffers, garantizando interfaces bien definidas y con tipado estricto entre los servicios.

- **Eventos asíncronos (Kafka):** Para desacoplar procesos y lograr mayor flexibilidad, el sistema emplea un bus de mensajería (por ejemplo, Apache Kafka) donde los servicios publican eventos y otros servicios los consumen de forma asíncrona. Esta comunicación orientada a eventos permite que, tras ciertas acciones, se notifique a múltiples interesados sin necesidad de llamadas directas. Por ejemplo, cuando se completa una transferencia P2P, el **payments-service** podría emitir un evento "TransferCompleted" en un tópico de Kafka. Otros servicios podrían estar suscritos: uno de notificaciones enviará un recibo o alerta al usuario, otro de analítica contabilizará la transacción para fines estadísticos, etc., todo esto sin impactar el flujo principal de la transferencia. De modo similar, tras el onboarding y verificación KYC de

un cliente, el **account-service** podría publicar un evento "CustomerVerified", que el **ledger-service** podría consumir para automáticamente crear la cuenta contable correspondiente (alternativa al enfoque sincrónico de invocar gRPC). Gracias a Kafka, estas interacciones no requieren que los servicios estén fuertemente acoplados o activos al mismo tiempo: el envío y recepción de mensajes es tolerante a fallos temporales y facilita escalar cada componente según la carga de eventos.

En combinación, ambos mecanismos (gRPC y eventos) aseguran una arquitectura híbrida: consistente y rápida para las operaciones en línea, y a la vez flexible y extensible mediante procesos en segundo plano. Este diseño mejora la robustez general del sistema y permite integrar nuevos servicios en el futuro simplemente suscribiéndolos a los eventos relevantes o exponiendo nuevas RPCs según convenga.

Detalles Técnicos y Principios de Diseño

A continuación, se profundiza en aspectos técnicos de la implementación y en los principios de diseño que guían el proyecto, incluyendo tecnologías usadas, organización del código, persistencia y observabilidad del sistema:

Stack Tecnológico Utilizado

- **Lenguaje Go:** Todos los microservicios están escritos en Go (Golang) por su eficiencia y sencillez para construir servicios concurrentes. Go ofrece un excelente soporte para construir servidores gRPC de alto rendimiento y con bajo consumo de recursos.

- **gRPC & Protocol Buffers:** La comunicación interna se define con Protocol Buffers (archivo ".proto") y se implementa con gRPC. Esto proporciona interfaces bien definidas para cada servicio y genera stubs de cliente/servidor en Go automáticamente. Por ejemplo, el **ledger-service** define en su proto métodos como CreateAccount, PostTransfer y GetBalanceCents, que otros servicios invocan a través de gRPC.

- **PostgreSQL:** Cada microservicio persiste sus datos en una base de datos relacional PostgreSQL independiente. Esto asegura consistencia transaccional y un esquema por dominio. Por ejemplo, el **ledger-service** se conecta a una base PostgreSQL (configurada mediante variables de entorno) donde almacena sus tablas de cuentas y transacciones. La elección de Postgres permite usar SQL estándar, integridad referencial y facilidad para escalar en volumen de datos financieros.

- **Kafka:** El sistema utiliza Apache Kafka como plataforma de mensajería para un diseño orientado a eventos. Kafka actúa como broker donde se publican eventos (en tópicos específicos) que pueden ser consumidos por uno o varios servicios suscriptores. Su alta capacidad de procesamiento de mensajes en tiempo real y persistencia lo hacen ideal para las necesidades de un core bancario event-driven (por ejemplo, para registrar eventos de auditoría, notificaciones, sincronización con otros sistemas, etc.).

- **Docker & Contenedores:** Para estandarizar el despliegue, los servicios y componentes como la base de datos se dockerizan. Existe, por ejemplo, un docker-compose.yml que

levanta instancias de Postgres para desarrollo. Cada microservicio puede empaquetarse en una imagen Docker, facilitando la puesta en marcha del entorno local y la eventual orquestación en ambientes de producción (Kubernetes u otra plataforma de contenedores).

- **Bibliotecas y utilidades:** Se aprovechan librerías de la comunidad Go para tareas comunes. Por ejemplo, se utiliza Zap (Uber) para logging estructurado de alta performance, la librería github.com/lib/pq como driver de Postgres, y herramientas como migrate para manejar versiones de esquema de base de datos (aplicación de migraciones SQL). No se incluyen frameworks pesados; el enfoque se mantiene ligero, apoyándose en el propio estándar de Go y pequeñas librerías especializadas.

Principios de Diseño Clave

- **Separación de dominios (Single Responsibility):** Cada microservicio corresponde a un dominio de negocio claramente definido y encapsula toda la lógica y datos de ese ámbito. Por ejemplo, el **ledger-service** solo se ocupa de la lógica contable y no "sabe" nada de clientes personales, mientras que el **account-service** maneja los datos del cliente y reglas de onboarding sin involucrarse en cómo se registran transacciones financieras. Esta separación de responsabilidades facilita el desarrollo independiente de cada componente y minimiza el impacto de los cambios (p.ej., modificar algo de KYC no afectará el módulo de pagos). También implica que la base de datos está aislada por servicio – no hay un esquema global monolítico sino varios esquemas/bases menores, uno por dominio, evitando acoplamiento a nivel de datos.

- **Diseño orientado a eventos (Event-Driven):** El sistema adopta una arquitectura reactiva donde los servicios emiten y responden a eventos para ciertas acciones. En lugar de depender únicamente de peticiones sincrónicas directas, muchos flujos se desacoplan usando mensajería: un servicio publica un evento cuando ocurre algo importante y otros reaccionan en consecuencia. Esto introduce elasticidad y extensibilidad en el diseño: nuevos requisitos se pueden implementar añadiendo suscriptores a eventos existentes en lugar de modificar los servicios emisores. Además, se mejora la robustez, ya que una falla en un servicio consumidor no bloquea la operación principal (el evento queda en cola en Kafka hasta que pueda ser procesado). El proyecto sigue este principio para cosas como notificaciones, creación de cuentas tras KYC, logging de auditoría, etc., asegurando que el núcleo de cada servicio se mantenga simple y delegando tareas adicionales vía eventos.

- **Consistencia e idempotencia:** Dado el carácter financiero del sistema, se pone énfasis en garantizar la consistencia de los datos y evitar duplicidades en las operaciones. Todas las operaciones críticas (ej: registrar una transferencia) se envuelven en transacciones de base de datos, de modo que ante un fallo parcial no queden datos inconsistentes. Asimismo, se implementan mecanismos de idempotencia: por ejemplo, cada transferencia P2P lleva un identificador externo único para que, si por error se envía dos veces la misma orden, el **ledger-service** la rechace por duplicada . Esto previene duplicar movimientos de dinero ante reintentos o

problemas de comunicación. Otro ejemplo es la verificación de balance cero en cada transferencia en **ledger-service**, garantizando que los débitos y créditos se compensan correctamente. Estos principios aseguran integridad financiera en todo momento.

- **Escalabilidad horizontal:** Al estar dividido en microservicios, el sistema puede escalar de forma horizontal según la carga de trabajo de cada componente. Si el número de transacciones crece enormemente, se pueden desplegar instancias adicionales de **payments-service** o **ledger-service** detrás de un balanceador, sin necesidad de escalar todo el sistema monolíticamente. Del mismo modo, Kafka permite manejar picos de eventos, y la base de datos de cada servicio puede escalarse (usando réplicas, sharding, etc.) independientemente. La separación de dominios combinada con comunicación vía eventos hace más sencilla la adaptación a grandes volúmenes o a la distribución geográfica de la carga.

Persistencia de Datos y Modelo de Información

Cada servicio administra su propia persistencia de datos, lo que significa que mantiene un almacén dedicado (generalmente una base de datos SQL) con las entidades relevantes de su dominio. Esta estrategia de base de datos por microservicio refuerza el encapsulamiento y la seguridad de los datos. En Neobank se optó por bases de datos relacionales (PostgreSQL) debido a la naturaleza transaccional del negocio financiero:

- El **ledger-service** utiliza un esquema relacional para almacenar: cuentas contables (Ledger accounts), transfers (Transfers) y asientos (Entries). Aprovecha transacciones SQL para las operaciones de inserción de transferencias y asientos, asegurando commit/rollback automático en bloque. Los saldos no se almacenan materializados, sino que se calculan agregando las entradas en cualquier momento, manteniendo la fuente de verdad en los asientos (esto evita inconsistencias, a costa de requerir consultas sumatorias, aunque se podrían agregar índices o tablas de balance para optimización más adelante).

- El **account-service** contaría con su propia base de datos, probablemente con tablas como customers (clientes), kyc_requests (documentos o info de verificación) y quizás accounts (que vinculen un cliente con su(s) cuenta(s) contable(s) del ledger, almacenando el ID de cuenta ledger retornado). De esta manera, cuando un cliente nuevo es verificado, el **account-service** guarda la relación entre el cliente y la cuenta contable creada.

- El **payments-service** podría no necesitar almacenar mucha información permanente (ya que delega a **ledger-service** el registro de transferencias). Aun así, podría llevar un registro de historial de pagos solicitados o estados temporales (especialmente si hubiera funcionalidades como pagos programados, reintentos, etc.). Cualquier dato que persista sería en un almacén propio, independiente de los demás.

Todos los accesos a datos se realizan a través de las interfaces de los servicios. No hay consultas directas de un servicio a la base de otro. Por ejemplo, si el **account-service** requiere

saber el saldo de una cuenta, en lugar de leer la base del ledger (lo cual violaría la segregación), invoca al método provisto por **ledger-service** (GetBalanceCents) y espera su respuesta. Este aislamiento previene efectos colaterales y permite incluso que cada base de datos pueda tener configuraciones o tecnologías distintas si en un futuro algún dominio lo necesitara (p.ej., podríamos decidir usar una base documental para almacenar documentos de identidad escaneados, sin afectar a los otros servicios).

En cuanto al modelo de datos compartido, se reduce al mínimo indispensable: típicamente los servicios intercambian solo identificadores y datos necesarios a través de las APIs. Por ejemplo, **ledger-service** solo maneja IDs de cuenta contable y montos, sin conocer detalles del usuario; el **account-service** traduce las operaciones de un usuario a llamadas con los IDs internos correspondientes. Esta división sigue los lineamientos de Domain-Driven Design, evitando que un servicio tenga conocimiento innecesario de la lógica interna de otro.

Observabilidad y Métricas

Para asegurar que el sistema sea mantenible y que los desarrolladores puedan diagnosticar problemas fácilmente, Neobank incorpora prácticas de observabilidad, abarcando logging, métricas de monitoreo y (en el futuro) trazas distribuidas:

- **Logging estructurado:** Cada microservicio genera logs detallados de sus operaciones usando la librería Zap de Uber, configurada en modo de desarrollo para legibilidad. Los logs incluyen contexto estructurado (ej. identificadores de solicitud, IDs de cuenta, montos) lo que facilita filtrarlos y analizarlos. Por ejemplo, el **ledger-service** registra eventos importantes como la inicialización del servicio, errores de conexión a la base de datos, creación de cuentas y resultados de transferencias (incluyendo un log en nivel debug cuando una cuenta es creada exitosamente). Estos logs pueden centralizarse mediante una herramienta de agregación (como Elastic Stack o Grafana Loki) para monitorear el sistema en tiempo real y hacer debugging post-mortem de incidencias.

- **Métricas de rendimiento y negocio:** El sistema expone métricas clave que permiten conocer su estado de salud y el uso de las funcionalidades. Si bien aún no se detalla en el código, la intención es integrar un sistema de métricas (por ejemplo, usando un cliente Prometheus en Go) en cada servicio. Esto recopilaría datos como: número de solicitudes procesadas por segundo en cada servicio, latencia promedio de llamadas gRPC, tasa de errores (por tipo de error), uso de CPU/memoria, etc. Adicionalmente, se medirán métricas de negocio importantes, tales como el conteo de clientes registrados, porcentaje de usuarios que pasan KYC exitosamente, número de cuentas contables activas, volumen diario de transferencias efectuadas y saldo total manejado por el sistema. Estas métricas de alto nivel ayudan a evaluar el crecimiento de la plataforma y detectar comportamientos anómalos (por ejemplo, una caída repentina en tasas de aprobación KYC podría indicar un problema en ese módulo). Todos estos datos serían recopilados y visualizados en dashboards (usando herramientas como Grafana), permitiendo al equipo de desarrollo y operaciones observar el funcionamiento interno del sistema de forma continua.

- **Trazabilidad de extremo a extremo:** En una arquitectura de microservicios, una sola acción de usuario (como realizar una transferencia) involucra múltiples hops entre servicios. Para facilitar seguir el rastro de una transacción a través de todo el sistema, Neobank contempla implementar tracing distribuido. Esto se podría lograr integrando herramientas como OpenTelemetry o similares, de forma que cada solicitud entrante lleve un identificador de traza que se propaga por las llamadas gRPC y eventos Kafka. Así, se podría reconstruir el camino completo de, por ejemplo, la transferencia #123: desde el request inicial en **account-service**, pasando por **payments-service**, hasta las operaciones en **ledger-service** y cualquier evento emitido. Esto brinda una poderosa capacidad de diagnosticar cuellos de botella o fallos, entendiendo exactamente dónde pudo haber demoras o errores en la cadena de servicios. Si bien su implementación completa es futura, el diseño actual ya considera ciertas piezas para facilitarlo (por ejemplo, el uso del externalID en transfers actúa como identificador común en logs de diferentes servicios para una misma operación).

En conjunto, estas estrategias de observabilidad y monitoreo garantizan que el sistema no sea una “caja negra”, sino que ofrezca transparencia tanto en tiempo real como retrospectiva. Un desarrollador nuevo puede apoyarse en logs y métricas para entender el comportamiento de los servicios, y el equipo puede detectar incidentes o degradaciones antes de que afecten gravemente a los usuarios finales.

Conclusión

En esta documentación se presentó una visión completa del proyecto Neobank, cubriendo desde una introducción para nuevos desarrolladores, los flujos funcionales esenciales, hasta la arquitectura de microservicios que sustenta el sistema y los detalles técnicos clave. En resumen, Neobank es un sistema de banca digital modular, donde cada microservicio desempeña un rol específico (clientes, cuentas contables, pagos) y colaboran mediante gRPC y eventos para ofrecer funcionalidades de onboarding de usuarios, cumplimiento KYC y transferencias P2P de manera consistente y escalable. La elección de tecnologías (Go, gRPC, PostgreSQL, Kafka) y la adopción de principios de diseño sólidos (separación de dominios, diseño orientado a eventos, observabilidad) buscan lograr un sistema robusto que sea fácil de entender, extender y mantener. Cualquier desarrollador que se incorpore al proyecto podrá enfocarse en un servicio a la vez, con la confianza de que hay contratos claros entre componentes y mecanismos para monitorear y depurar el comportamiento del sistema en su conjunto. De esta forma, Neobank sienta las bases para evolucionar hacia un producto financiero completo, manteniendo la calidad y la fiabilidad en cada paso del camino.