HOW WOULD YOU DEPLOY THE SYSTEM ABOVE? IT NEEDS TO MEET THE FOLLOWING REQUIREMENTS:

(1) UPDATES TO BUYERS NEED TO BE DELIVERED AT LEAST ONCE.

(2) LATENCY BETWEEN RECEIVING AN UPDATE AND SENDING IT OUT TO BUYERS SHOULD BE AS LOW AS POSSIBLE.

To meet the requirements of ensuring **updates to buyers are delivered at least once** while keeping **latency between receiving and sending updates as low as possible**, a robust system architecture is essential.

Delivering Updates to Buyers at Least Once

In systems that handle frequent real-time updates—such as notifying buyers about deal changes, price fluctuations, or new bid statuses—**message reliability** is of paramount importance. The first requirement is to guarantee that every buyer receives their updates at least once, even in the face of possible failures.

To achieve this, we can employ **asynchronous messaging** mechanisms with **message queues** like **RabbitMQ** or **Apache Kafka**. These systems allow you to publish messages to a queue whenever a deal update occurs. Buyers (or a notification service acting on their behalf) subscribe to these messages. By storing and processing these messages asynchronously, the platform decouples the act of sending updates from the act of processing them. This design ensures that messages are handled reliably, as the queue can store them until they are acknowledged, guaranteeing delivery even if there are network outages or server failures.

For example, when a deal is updated or a price change happens, the system immediately pushes a message into a **queue**. Buyers listening to updates for that deal can consume the message and act on it - such as displaying the change on their user interface or sending a notification to the buyer. If a buyer is offline or the server crashes during message processing, the message will remain in the queue until it is successfully delivered. This **at-least-once delivery guarantee** ensures no important update is lost.

To further strengthen message reliability, **deduplication techniques** should be employed on the consumer side. This ensures that even if a message is delivered more than once (a common scenario in at-least-once delivery systems), it will not cause unwanted side effects like duplicate notifications.

Reducing Latency Between Update and Notification

Minimizing latency is crucial to deliver real-time updates to buyers. To achieve this, the platform can adopt an **event-driven architecture** where events are processed as soon as they occur.

For example, as soon as a deal is modified, an event is published to the message queue, and buyers are notified immediately. The use of **WebSockets** or **Server-Sent Events (SSE)** allows for persistent, real-time communication channels between the server and the client. These technologies ensure that as soon as the system processes an event, the buyer receives it instantly without the delay of polling for updates.

To further reduce latency, **data caching** with a system like **Redis** can be used to store frequently accessed information, such as active deals. By serving cached data, the system avoids repeated database queries, allowing for faster responses to buyer requests. Additionally, deploying the application across **geographically distributed servers** or utilizing **edge computing** ensures that users receive updates from servers closest to them, reducing the time taken for messages to travel across the network.

IN THE FUTURE, THIS PLATFORM WOULD NEED TO BE ABLE TO ACCEPT BIDS FROM BUYERS AND SEND THEM INVOICES UPON SALE END. HOW WOULD YOU IMPLEMENT IT?

To enable the platform to accept bids from buyers we can make use of **(We already have this in our current system schema)** / introduce database schemas like the ones below for Bid and Invoices

```
model Bid {
 id        String    @id @default(uuid())
 amount    Float
 buyer     Buyer     @relation(fields: [buyerId], references: [id])
 buyerId   String
 deal      Deal      @relation(fields: [dealId], references: [id])
 dealId    String
 createdAt DateTime  @default(now())
 updatedAt DateTime  @default(now())
 deletedAt DateTime?
}
```

```
// Invoice model representing invoices for deals with soft delete
model Invoice {
 id            String    @id @default(uuid())
 deal          Deal      @relation(fields: [dealId], references: [id])
 dealId        String    @unique
 buyer         Buyer     @relation(fields: [buyerId], references: [id])
 buyerId       String
 totalAmount   Float
 paymentMethod String?
```

```
  paidAt         DateTime?
  createdAt      DateTime  @default(now())
  updatedAt      DateTime  @updatedAt
  deletedAt      DateTime?
}
```

Bid Entity Relationships:

- **Buyer**: Each bid is associated with a buyer, linking to the Users table via buyerId.
- **Deal**: Each bid is linked to a specific deal in the Deals table via dealId.

Buyer Entity Relationships

- **Buyer**: Each invoice is associated with a buyer, linking to the Users table via buyerId.
- **Deal**: Each invoice is linked to a specific deal in the Deals table via dealId.

Bid and Invoice Implementation Strategy

- Create a Bid Management Service that handles bids, including submission, status updates, and notifications.
- Implement real-time notifications for buyers when their bid status changes.
- Develop an Invoice Generation Service to create invoices upon deal completion.
- Integrate with payment gateways to handle transactions.
- Use email or SMS notifications to inform buyers about bid outcomes and invoice generation.
- Implement webhook functionality to receive payment confirmations from payment providers.
- Design services to scale independently based on user activity, allowing for efficient handling of increased bidding activity during peak times.
- Ensure that indices and relationships are optimized for fast querying as bid and invoice volumes grow.

Workflow for Bids and Invoicing on the Trading Platform

Buyers browse available deals and decide to place a bid on a deal they are interested in. The buyer submits a bid, which is processed by the Bid Management Service. The bid includes details such as the buyer's ID, the deal ID, the bid amount, and a timestamp. The new bid is recorded in the Bid Table, updating its status to "Pending."

Sellers are notified of new bids on their deals. This can be implemented using real-time notifications via WebSockets or server-sent events. The seller reviews the bids and decides whether to accept or reject each one. The seller updates the status of the bid (Accepted or Rejected), and this change is reflected in the Bid Table. The buyer is notified of the bid's outcome through email or SMS.

Once a bid is accepted, the Invoice Generation Service creates an invoice. The invoice includes the buyer's details, deal information, total amount, and payment status. The invoice is stored in the Invoice Table, linking it to the buyer and the deal.

The platform integrates with a payment gateway (like Stripe or PayPal) to process payments for accepted bids. Upon successful payment, the invoice status is updated to "Paid." The payment provider sends a webhook to the platform, confirming the payment, which can trigger further actions, such as notifying the seller.

All bid and invoice activities are logged for auditing and reporting purposes, ensuring transparency and accountability in the bidding and invoicing process. The system also provides support for any issues that arise regarding bids or invoices, facilitating smooth user interactions and enhancing overall user experience.

WHAT OTHER CHALLENGES DO YOU SEE WITH THIS SYSTEM? HOW WOULD YOU APPROACH THEM?

Some of the challenges this system could face **include database connection issues**, **scalability concerns, security and compliance requirements, real-time features, and performance under load.**

To address database connection issues, implementing a connection pooling mechanism can help manage database connections efficiently. Building robust error handling will allow the system to manage connection timeouts and retries gracefully, ensuring that users receive meaningful feedback when issues arise.

For scalability, we can introduce microservices to separate concerns, such as bidding, invoicing, and user management, allowing each service to scale independently. Utilizing container orchestration tools like Kubernetes will help manage service scaling based on real-time traffic patterns.

Regarding security and compliance, it’s vital to use SSL/TLS for data in transit and to encrypt sensitive information stored in the database. Conducting regular audits and vulnerability assessments will ensure compliance with industry standards and help protect against potential threats.

In terms of real-time features, integrating an event-driven architecture using a message queue, such as RabbitMQ or Bull, can help manage and distribute notifications without negatively impacting user experience. Additionally, employing an efficient WebSocket solution will facilitate real-time communications while minimizing server overload.

For performance under load, implementing caching strategies with technologies like Redis will store frequently accessed data and reduce database load. Regular load testing should be conducted to identify bottlenecks and ensure the system can handle expected traffic.

Lastly, for scalability for future features, breaking the application into microservices will allow individual components, like bidding or invoicing, to scale independently based on demand. This approach not only prepares the system for future growth but also enhances its overall resilience and robustness.