---

1. The Core Problem: The "Final Second" Race Condition

The primary risk is **Double-Spending** or **Stale-State Bidding**.

- **User A and B** both see a bid of **$100**.
- Both click "Bid +$10" simultaneously.
- In a naive system, both bids are processed as **$110**. User A wins, but User B *thinks* they won too.
- **Latency Spikes:** As users increase ($100 \rightarrow 1000$), traditional DB locking creates a bottleneck. If the DB takes 500ms to lock a row, the "Real-Time" experience is ruined.

---

2. High-Level Architecture (The "Buffer & Stream" Approach)

To reduce DB calls and latency, we use an **In-Memory Authority (Redis)**. The Database becomes the *archival* layer, while Redis handles the *transactional* layer.

The Components:

1. **Client (React):** Calculates local time offset from the server to keep the countdown accurate without constant polling.
2. **Socket Layer (Node.js):** Stateless workers. They don't store data; they just talk to Redis.
3. **The Hot Layer (Redis):** Stores active auction state. We use **Atomic Operations (LUA Scripts)** to ensure that "Compare-and-Swap" happens in one CPU cycle.
4. **The Persistence Layer (PostgreSQL):** Every 5 minutes (or via a Message Queue), the winning state is synced to the DB.

---

3. Latency & Scalability Projections

The goal is to keep "Glass-to-Glass" latency (User A clicks $\rightarrow$ User B sees update) under **200ms**.

User Count	Expected Latency	Bottleneck	Mitigation Strategy
100 Users	**~20ms**	Network RTT	None needed.
1,000 Users	**~50ms**	Socket Fan-out	Use Redis Pub/Sub to distribute updates across multiple Node.js instances.
10,000 Users	**~150ms**	CPU context switching	Vertical scaling of Node.js instances + Load Balancer.
100k+ Users	**>1s (Potential)**	Network Bandwidth	Implement **Delta Updates** (send only the price, not the whole object).

---

4. Approach: The "Atomic Lua" Write-Back Design

To fulfill your request for a "price-saving" architecture that reduces DB calls, we implement a **Write-Through Cache with a Deferred Sink**.

Step 1: The Atomic Bid (Lua Script)

Instead of GET price -> Logic -> SET price, we send a script to Redis. Redis executes this as a single atomic unit. No two bids can overlap.

```lua
-- Redis LUA Script
local current_bid = tonumber(redis.call('get', KEYS[1]))
local new_bid = tonumber(ARGV[1])
local user_id = ARGV[2]

if new_bid > current_bid then
    redis.call('set', KEYS[1], new_bid)
    redis.call('set', KEYS[1] .. ':user', user_id)
    return 1 -- Success
else
    return 0 -- Outbid
end
```

Step 2: Reducing DB Calls (The Sync Loop)

We don't write to the DB on every bid. We use a **Stream/Queue**.

1. Every successful bid is pushed into a **Redis Stream**.
2. A background worker (Cron or Worker Thread) "drains" the stream every 60 seconds (or when it hits 100 items) and performs a BULK INSERT into PostgreSQL.
3. This reduces DB I/O by **90-95%**.

---

5. Technical Implementation Details

A. The "Anti-Hack" Timer Synchronization

- **Server:** Sends serverTime and endTime in the initial payload.
- **Client:** Calculates offset = serverTime - Date.now().
- **The Countdown:** const remaining = endTime - (Date.now() + offset).
- This ensures that even if the user changes their system clock, the countdown remains pinned to server time.

B. "Senior" Dockerfile Strategy

We use a multi-stage build to keep the production image lean (approx. 100MB).

Dockerfile

```
# Stage 1: Build
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2: Production
FROM node:20-alpine
WORKDIR /app
ENV NODE_ENV=production
# Only copy necessary files (Cost & Security optimization)
COPY --from=builder /app/dist ./dist
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/package.json ./

USER node
EXPOSE 3000
CMD ["node", "dist/server.js"]
```