**Simple web-based chat application:**

**Technologies used:**

1. NodeJS as backend - Event driven, non-blocking I/O model, suitable for real time messaging.
2. Socket.io Real time bidirectional communication between two nodes
3. Redis for faster in-memory caching to store the sticky session information.
4. Load balancer built on top of Amazon EC2 instances.
5. MySQL database for persistent chat history storing.

**Architecture setup:**

* A client could be web/mobile that supports Socket.io client-side library
* Servers (EC2 instances) supports sticky sessions (temporarily), which means that for certain time during the chat, a client can be attached to a server instance and that information is maintained by Redis caching.
* MySQL database stores persistent chat history of clients in parallel.
* Load balancer to assign client to the server instances that are available.

**Messaging scenario:**

* Client A sends a message to another client B, along with both the A and B client ID.
* Load balancer checks in Redis if client A is already assigned to any server. If not, it assigns a server to the client A and this is stored in Redis memory. The same happens for client B.
* Now the server for client A checks where the client B is assigned to, by making a READ operation on Redis, where the message is passed through.
* In addition to this, a WRITE operation is done on the MySQL DB about this chat instance.
* When B responds back, the same process of querying Redis and MySQL chat data storing happens.
* If any client wants to retrieve chat history, then MySQL DB is queried for it and it will be sent back using REST protocol (No need to use Socket communication)

**Coding:**

* I would go about coding the frontend in React, with Redux to manage the state, using TypeScript, TDD to avoid bugs.
* For backend, ExpressJS will be used, in TypeScript.
* For ORM, Sequelize can be used.
* Will use Docker for ease of deployment without any worries for missing dependencies.

**What’s more?**

* Same architecture can be extended for group messaging (one client to a server TO many clients as a cluster, connected to a server).
* Online status, received and read receipts using another instance of Redis.