BUILDING A HIGH-PERFORMANCE REAL-TIME CHAT SYSTEM USING MSSQL: A SCALABLE, AI-DRIVEN ALTERNATIVE TO PROPRIETARY CHAT SOLUTIONS

Sipilov I.

Sipilov Ivan — Head of Platform Development,
CITADEL HEDGE FUND,
FOUNDER OF NANNYSERVICES.CS,
NEW YORK, USA

Abstract: *this paper explores the design and implementation of a custom-built, real-time chat system using MSSQL as a backend database. Serving over 480,000 active users globally and processing more than 4 million messages daily, the system was developed to be highly performant, cost-efficient, and fully self-hosted, without reliance on external tools such as Firebase. Additionally, the system integrates a GPT-based post-moderation model to automatically detect and flag problematic messages, such as those involving harassment or abuse. The model learns and improves using machine learning based on user moderation flags, enhancing future detection. This article focuses on the technical implementation, scalability, and performance, alongside the ethical considerations of using AI-driven moderation.*

Keywords: *Real-time chat system, MSSQL, scalable chat platform, AI-driven moderation, GPT-based moderation, custom-built chat solution, machine learning for moderation, SignalR real-time communication, in-memory OLTP, Service Broker, Change Data Capture, horizontal scaling, reactive event store, Firebase alternative, self-hosted chat platform, cost-efficient chat system, privacy in chat systems, global chat infrastructure, harassment detection, message flagging, vendor lock-in, relational database in real-time systems, message throughput, message partitioning, ethical AI moderation, data sovereignty in chat platforms, high-volume messaging system.*

1. Introduction

- **Context**: Real-time chat systems are essential for modern communication, especially for organizations needing instant messaging capabilities with full control over their infrastructure. Solutions such as Firebase offer ready-to-use frameworks, but they come with downsides such as high costs, vendor lock-in, and limited flexibility. For larger platforms processing millions of messages daily, these solutions may not be optimal.
- **Objective**: This paper details the implementation of a custom chat system powered by MSSQL, which handles over **4 million messages daily** and supports **480,000 active users globally**. The system is entirely self-hosted, offering full control over data and infrastructure. In addition, a GPT-based post-moderation system is integrated to automatically flag and address problematic content.
- **Significance**: By avoiding commercial frameworks and creating a tailored solution, the system ensures cost savings, complete data sovereignty, and the ability to customize functionality to meet specific user needs.

2. Related Work

- **Existing Chat Solutions**: Commercial offerings like Firebase and Twilio provide real-time chat capabilities but can be expensive at scale and limit customization due to proprietary architectures. There has been increasing interest in self-hosted alternatives, particularly for organizations prioritizing privacy and cost-effectiveness.
- **Real-Time Databases**: Relational databases like MSSQL have seen significant advances, making them capable of handling real-time workloads typically managed by NoSQL databases. Features such as **Service Broker**, **Change Data Capture (CDC)**, and **in-memory tables** position MSSQL as a viable backend for real-time applications.
- **AI-Based Moderation**: Moderation using AI models such as **GPT** has become increasingly relevant in chat systems, particularly with the rising concern over harassment, abuse, and toxic behavior. These models analyze the content of messages and apply filtering based on both user-defined rules and machine learning techniques.

3. System Architecture

3.1 System Overview

The chat system is designed to be a scalable, high-performance platform that integrates an MSSQL backend and machine learning-based moderation. The system handles over 4 million messages daily across 480,000 active users distributed globally.

- **Backend Server**: A lightweight, efficient backend using **ASP.NET Core** handles message routing, real-time notification, and connection management.
- **MSSQL Database**: The database manages message storage, near-real-time updates, and indexing, ensuring rapid message retrieval even at high volumes. Low data access latency is achieved by storing recent portions of data

in **memory-optimized tables**. That is, these structures are designed to keep records in memory rather than storing it on the disk. This feature also removes lock and latch contention between concurrently executing transactions.

- ~~**Real-Time Features**: The MSSQL system uses **Service Broker** for real-time messaging and **CDC** to track changes in real-time for notifications, reducing latency and enabling rapid message processing.~~
- **AI-Based Moderation**: A **GPT-based model** continuously monitors message content, identifying problematic phrases and behaviors such as harassment or abuse. The model learns over time through user feedback and moderation flags.

4. Implementation and Technical Details

4.1 Real-Time Messaging System

- **Message Storage**: Messages are stored in a centralized **memory optimized Message Table** that supports rapid access and retrieval. Key fields include:
  - **Message ID** (Primary Key)
  - **Sender and Receiver IDs**
  - **Timestamp**
  - **Content** (Text of the message)
  - **Status** (Unread, Read, Flagged)
- **Reactive event store:** Using internal event-driven reactive in-memory implementation of event store, system enables instantaneous message delivery. Upon message creation, an event is pushed to this store. Listeners subscribe to the store for new event notifications implementing “observable” pattern. When such notification arrives, the backend server immediately routes state updates to the recipient.

Example code of subscribing to event store events inside the SignalR hub method (simplified for readability). Here the subscription is being made for events that could change current chat state for a particular user. When such an event is raised, the new slice of chat state is read from the database in **GetChatStateAsync** method and is automatically pushed to the channel associated with the current user’s connection.

```
public ChannelReader<ChatStateViewModel> GetChatStateStream()
{
  var customerId = GetCustomerId();
  var initialChatStateObservable = Observable.FromAsync(GetChatStateAsync)
     .Replay(1)
     .RefCount();

  var onEventChatStateObservable = initialChatStateObservable.SelectMany(initialChatState =>
  {
     var messageSentEvents = _events.GetEvent<MessageSentEvent>()
        .Where(@event => @event.SenderId == customerId);
     var messageReceivedEvents = _events.GetEvent<MessageReceivedEvent>()
        .Where(@event => @event.ReceiverId == customerId);
     var conversationChangedEvents = _events.GetEvent<ConversationChangedEvent>()
        .Where(@event => @event.ConversationId.GetConversationUserIds()
           .Contains(customerId));
     var sendingBlockedForUserEvents = _events.GetEvent<SendingBlockedForUserEvent>()
        .Where(@event => @event.CustomerId == customerId);
     var messageDeletedEvents = _events.GetEvent<MessageDeletedEvent>()
        .Where(@event => @event.ReceiverId == customerId || @event.SenderId == customerId);
     var messageModerationStatusChangedEvents =
_events.GetEvent<MessageModerationStatusChangedEvent>()
        .Where(@event => @event.ReceiverId == customerId || @event.SenderId == customerId);
     return messageSentEvents.Cast<IEvent>()
        .Merge(messageReceivedEvents)
        .Merge(conversationChangedEvents)
        .Merge(sendingBlockedForUserEvents)
        .Merge(messageDeletedEvents)
        .Merge(messageModerationStatusChangedEvents)
        .SelectMany(_ => Observable.FromAsync(GetChatStateAsync)
           .Where(state => state != null));
  });
```

```
    var result = initialChatStateObservable.Merge(onEventChatStateObservable);
    return result.AsChannelReader(out _subscription);
  }
```

- **Real-Time Notifications:** Backend server uses **ASP.NET Core SignalR** library that enables server-side code to push content to clients instantly. Under the hood, SignalR uses the following techniques for handling real-time communication with clients (in order of graceful fallback):
- WebSockets
- Server-Sent Events
- Long Polling
- **In-Memory Optimization**: To handle the high volume of messages (over 4 million per day), the system uses **in-memory OLTP** for fast access to recently exchanged messages. Messages older than a set threshold are archived in the disk-based primary storage for historical access.

Example SQL procedure for storing new message in DB (simplified for readability):

```
CREATE PROCEDURE [dbo].[SendChatMessage]
  @senderId uniqueidentifier,
  @receiverId uniqueidentifier,
  @text nvarchar(1024) = NULL,
  @createDate datetime2(7),
  @requireFirstMessagePreModeration bit,
  @messageType int = NULL,
  @payloadJson nvarchar(4000) = NULL,
  @conversationId nvarchar(450) OUTPUT,
  @messageId int OUTPUT,
  @isFirstConversationMessage bit OUTPUT,
  @preModerationStatus int OUTPUT
WITH NATIVE_COMPILATION, SCHEMABINDING, EXECUTE AS OWNER
AS
BEGIN ATOMIC
WITH (TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'English', DELAYED_DURABILITY = ON)
  EXECUTE [dbo].[EnsureChatUserExists] @customerId = @senderId
  EXECUTE [dbo].[EnsureChatUserExists] @customerId = @receiverId

  DECLARE @outgoingConversationExists bit = IIF([dbo].[IsUserConversationExists](@senderId, @receiverId) = 1, 1, 0)
  DECLARE @incomingConversationExists bit = IIF([dbo].[IsUserConversationExists](@receiverId, @senderId) = 1, 1, 0)

  SET @isFirstConversationMessage = IIF(@outgoingConversationExists = 0 AND @incomingConversationExists = 0, 1, 0)
  SET @preModerationStatus = IIF(@requireFirstMessagePreModeration = 1 AND @isFirstConversationMessage = 1, 1, 0)

  SET @conversationId = [dbo].[GetConversationId](@senderId, @receiverId);

  DECLARE @messageHiddenForReceiver bit;
  SET @messageHiddenForReceiver = IIF(@preModerationStatus NOT IN (0, 2), 1, 0)

  INSERT INTO [dbo].[ChatMessage]
  (
    [SenderId],
    [ReceiverId],
    [ConversationId],
    [Text],
    [DateTime],
    [Deleted],
    [PreModerationStatus],
    [Unread],
    [UnreadNotified],
    [MessageType],
    [PayloadJson],
    [ModerationStatus]
  )
  VALUES
  (
    @senderId,
    @receiverId,
    @conversationId,
    @text,
    @createDate,
    0, -- Deleted - false
    @preModerationStatus, -- PreModerationStatus - Unspecified
    1, -- Unread - true
    0, -- UnreadNotified - false
    @messageType,
    @payloadJson,
    0 -- ModerationStatus - Unspecified
  )

  SET @messageId = SCOPE_IDENTITY()

  IF (@outgoingConversationExists = 0)
  BEGIN
    EXECUTE [dbo].[CreateUserConversation]
        @conversationId,
        @senderId,
        @receiverId,
        0, -- Archived - false
        0, -- Pinned - false
        @messageId,
        0, -- UnreadMessageCount - 0
        @createDate,
        1, -- IsInitiatedByUser - true
        0, -- Hidden - false
        0 -- MetadataFlags - no flags
  END
  ELSE
  BEGIN
    UPDATE
        [dbo].[UserConversation]
    SET
        [LastMessageId] = @messageId,
        [Archived] = 0 -- Archived - false
    WHERE
        [UserId] = @senderId
        AND [SecondUserId] = @receiverId
  END

  IF (@incomingConversationExists = 0)
  BEGIN
```

```
        EXECUTE [dbo].[CreateUserConversation]
            @conversationId,
            @receiverId,
            @senderId,
            0, -- Archived - false
            0, -- Pinned - false
            @messageId,
            1, -- UnreadMessageCount - 1
            @createDate,
            0, -- IsInitiatedByUser - false
            @messageHiddenForReceiver,
            0 -- MetadataFlags
    END
    ELSE
    BEGIN
        UPDATE
            [dbo].[UserConversation]
        SET
            [LastMessageId] = @messageId,
            [UnreadMessageCount] = [UnreadMessageCount] + 1,
            [Hidden] = IIF([Hidden] = 0, 0, @messageHiddenForReceiver)
        WHERE
            [UserId] = @receiverId
            AND [SecondUserId] = @senderId
    END
END
GO
```

4.2 Scalability Features

- **Partitioning:** To handle the large user base (480,000 active users) and to leverage efficient resource consumption messages are partitioned across hot and cold tables. All new messages are stored in a "hot" memory-optimized table for fast access. On a periodic configurable basis the maintenance procedure moves old messages to the "cold" disk-based partitioned table for archive access.
- **Message Indexing**: The system employs indexing strategies such as a **clustered index on message IDs** and **non-clustered indexes on timestamps and on user IDs** to ensure fast message retrieval and sorting.
- **Horizontal Scaling**: The system architecture supports horizontal scaling, where additional servers can be added to manage more users and messages without impacting performance.

5. AI-Based Post-Moderation Using GPT

5.1 Moderation System Overview

To ensure a safe and inclusive environment, the chat system integrates a GPT-based model for post-message moderation. The model processes messages in real-time, identifying potentially harmful or abusive content.

- **Model Capabilities**: The GPT-based system is designed to flag messages that may contain harassment, abusive language, or inappropriate content based on preset rules and learned patterns.
- **Learning with User Moderation Flags**: The moderation model improves over time using machine learning. When users flag specific messages as problematic, these flags are used to retrain the model, helping it better identify and block similar content in the future.

5.2 Ethical Considerations

- **User Privacy**: The GPT moderation system is designed to strike a balance between content moderation and user privacy. Only flagged content is stored for training, ensuring that the majority of message content remains private.
- **Transparency**: The system provides transparency to users by notifying them when messages are flagged or blocked by the moderation system.

5.3 Model Training

- **Initial Dataset**: The GPT model was initially trained on a dataset of flagged content from various chat systems, allowing it to recognize common abusive patterns such as harassment, hate speech, and threats.

- **Real-Time Updates**: User moderation flags allow for dynamic updates to the model. When a flagged message is reviewed and confirmed, the message content and metadata are fed back into the system to improve future predictions.

5.4 System Accuracy and Performance

- **Detection Rate**: The GPT-based model has an **85% accuracy rate** in identifying problematic content, improving with continued user interaction and flagging.
- **Message Processing Time**: The model processes messages in near real-time, with an average delay of **50-100 milliseconds** before content is flagged for moderation.

6. Results and Performance Benchmarks

6.1 System Performance

- **Message Throughput**: The system successfully handles over **4 million messages per day**, with average delivery times of **50-100 milliseconds** for real-time messaging.
- **User Base**: With **480,000 active users**, the system operates globally, ensuring message delivery across various regions without noticeable latency differences.

6.2 AI Moderation Performance

- **Flagging Efficiency**: Over **90% of flagged messages** are reviewed by the moderation team within minutes, and the system's accuracy in flagging inappropriate content is **85%**, with a **false positive rate** of less than **5%**.
- **Machine Learning Improvements**: With user moderation feedback, the model's ability to detect problematic content has increased by **20%** over the past six months, reducing the number of user-reported incidents.

7. Challenges and Solutions

7.1 Handling Message Load

Scaling to handle over 4 million messages per day was a challenge. Optimizations such as **cold-hot partitioning** and **in-memory tables** were critical for reducing the load on individual database nodes.

7.2 Moderation Challenges

One of the initial difficulties was handling edge cases in message flagging. For example, sarcasm or ambiguous language posed challenges for the GPT model, resulting in a higher false-positive rate. We addressed this by incorporating user feedback to train the model on more complex conversational nuances.

7.3 Data Privacy

Balancing real-time moderation with user privacy was a critical design challenge. To address this, only flagged messages were stored for model training, and clear privacy notices were provided to users.

Comparisons with the FIrebase, or any other alternative.

1. Introduction

As real-time communication becomes increasingly important in modern applications, developers face a choice between using external, managed services like **Firebase** and building custom, internal systems using databases like **MSSQL**. While Firebase offers convenience and ease of use with its cloud-based real-time messaging services, an internal chat system using MSSQL's real-time capabilities can offer more flexibility, better cost control, and enhanced performance, especially for large-scale applications.

In this paper, we conduct a comprehensive comparison between Firebase's messaging services and a custom-built real-time chat system using MSSQL, focusing on architectural differences, performance, scalability, and cost.

2. Overview of Firebase Messaging Services

2.1 Firebase Real-Time Database Architecture

Firebase's **Real-Time Database** is a cloud-hosted NoSQL database that stores data in a JSON format and synchronizes data between users in real time. Each client is connected to a central database, and whenever a change is made (such as a new message being sent), the database triggers an update, which is automatically propagated to all connected clients.

- **Data Storage**: Firebase stores data as a JSON tree, meaning the structure is hierarchical. This can create challenges as the database grows, leading to complex queries when trying to access deep nodes in the tree structure.
- **Real-Time Synchronization**: Firebase excels in real-time synchronization. Any update to the database is instantly broadcast to all clients subscribed to the relevant data nodes, making it ideal for chat systems where live updates are crucial.
- **WebSocket Connections**: Firebase uses **WebSockets** for maintaining persistent, bi-directional connections between the client and the server. This allows for real-time messaging, as updates are pushed instantly from the server to the clients without polling.

3. Overview of MSSQL Real-Time Chat System

3.1 MSSQL Real-Time Database Architecture

Unlike Firebase, which is a NoSQL database, **MSSQL** is a relational database, traditionally used for handling structured, transactional data. However, advances in MSSQL, such as ~~**Service Broker**~~, ~~**Change Data Capture (CDC)**, and~~ **in-memory OLTP** have enabled it to support real-time applications, including chat systems.

- ~~**Service Broker**: MSSQL's **Service Broker** enables asynchronous messaging within the database engine. It allows you to send and receive messages between applications in real-time without constant polling. This ensures that messages are processed and delivered instantly, making it suitable for real-time communication.~~
- ~~**Change Data Capture (CDC)**: MSSQL's **CDC** feature captures changes (such as inserts, updates, and deletes) in real-time, allowing you to monitor and react to data changes in your chat system. Whenever a new message is inserted into the database, CDC captures this change and notifies the application to push updates to the users.~~
- **In-Memory OLTP**: MSSQL's **in-memory OLTP** improves performance by keeping frequently accessed tables (such as recent messages or active conversations) in memory rather than disk storage. This results in faster read and write operations, dramatically reducing latency.

3.2 Real-Time Messaging with MSSQL

Messages in the MSSQL-based system are written directly to the **Message Table** in the database. When a new message is inserted:

1. **Reactive event store** immediately publishes a notification to the server.
2. The server, using **SignalR**, pushes the message to the recipient in real time.
3. The message data is stored in an **in-memory table**, ensuring that both the write and subsequent read operations are completed in milliseconds.

By leveraging **application-level event store**, systems can achieve near real-time notification without the need for constant database polling, which is a common performance bottleneck in systems that rely on manual database polling.

4. Performance Comparison

4.1 Firebase Messaging Performance

Firebase's messaging service is optimized for real-time synchronization across devices, and its **WebSocket-based** communication ensures low-latency updates. However, as Firebase is a NoSQL database, its performance can degrade with increasing complexity of the data structure.

- **Latency**: Firebase generally delivers messages with a latency of **100-200 milliseconds**, which is fast enough for most chat applications. However, as the database grows, Firebase's hierarchical structure can introduce bottlenecks, especially for large, complex datasets.
- **Data Scaling**: Firebase's performance is largely dependent on the size and structure of the JSON tree. As the depth of the tree increases, retrieving specific nodes (such as older messages or archived chats) becomes slower. This can impact the speed at which messages are retrieved and displayed to the user.

4.2 MSSQL Messaging Performance

MSSQL, traditionally known for its transactional capabilities, has evolved to support real-time workloads with the addition of **in-memory OLTP**~~, **Service Broker**, and **CDC**~~.

- **Latency**: In the MSSQL-based system, real-time message delivery is achieved with latency as low as **50-100 milliseconds**, making it faster than Firebase for both sending and receiving messages.
- **Data Scaling**: MSSQL's relational structure is inherently more efficient for querying large datasets. The use of **in-memory tables** ensures that messages are delivered and retrieved nearly instantaneously, regardless of the dataset size. Additionally, **Reactive event store implementation** avoids the overhead of polling, leading to better scalability as the number of messages grows.

4.3 Throughput and Load Handling

- **Firebase**: Firebase is designed to handle a large number of concurrent users, but as the user base grows, costs can become prohibitive. Additionally, as the number of concurrent messages increases, Firebase can experience performance degradation due to the inherent limitations of its JSON tree structure.
- **MSSQL**: MSSQL is capable of handling **4 million messages daily** across a global user base with **480,000 active users**. The use of **horizontal partitioning** and **sharding** ensures that the database scales effectively as the number of users grows. Messages are distributed across different database nodes based on **User ID** or **Conversation ID**, allowing for efficient load balancing and high throughput.

5. Cost Comparison

5.1 Firebase Pricing Model

Firebase follows a pay-as-you-go pricing model, where costs scale based on the number of **simultaneous connections**, **data storage**, and **read/write operations**. While Firebase is cost-effective for small-scale applications, it can become prohibitively expensive for large-scale systems, especially when handling millions of messages daily.

- **Connections**: Firebase charges for concurrent connections, which can quickly become costly as the number of users grows. For example, supporting **480,000 active users** could lead to significant monthly charges.
- **Storage and Bandwidth**: Firebase charges for data storage and bandwidth, meaning that high-message traffic applications like chat systems can incur substantial ongoing costs.

5.2 MSSQL Cost Model

Building a real-time chat system with MSSQL avoids the recurring costs associated with Firebase. While there are upfront infrastructure and licensing costs for MSSQL (depending on the SQL Server edition), the system allows for **complete control** over costs, and expenses do not scale linearly with the number of users or messages.

- **Licensing**: Depending on the edition of MSSQL being used (e.g., **Standard** or **Enterprise**), licensing fees may apply, but these costs remain predictable and avoid the exponential scaling seen in Firebase's pricing model.
- **Infrastructure**: By hosting the database internally or on a private cloud, organizations have more control over the infrastructure and can optimize the system to handle a large number of concurrent users without incurring additional costs.

6. Data Privacy and Control

6.1 Firebase Privacy Considerations

Firebase, being a Google product, stores all data in Google's cloud infrastructure. For organizations with strict **data privacy** and **data sovereignty** requirements, this may present a challenge. While Firebase offers secure storage and transmission of data, organizations have limited control over how their data is handled once it leaves their servers.

- **Vendor Lock-In**: Since Firebase is a proprietary service, migrating away from it can be challenging and time-consuming. Organizations are also tied to the terms and conditions of Firebase's service, which may not be suitable for all use cases.

6.2 MSSQL Privacy Considerations

With MSSQL, organizations retain complete control over their data. The database can be hosted on-premises or in a private cloud, ensuring full compliance with data sovereignty regulations such as **GDPR** and **HIPAA**.

- **Data Ownership**: All data remains within the organization's infrastructure, providing full control over how it is stored, accessed, and managed. This is particularly important for companies handling sensitive information, such as financial data or user communications.

7. Flexibility and Customization

7.1 Firebase Flexibility

Firebase provides a range of out-of-the-box functionality for real-time communication but is limited in terms of customization. Developers must work within Firebase's predefined architecture, which can make it difficult to implement custom features or optimizations that fall outside of Firebase's scope.

- **Custom Logic**: While Firebase allows the use of cloud functions to implement some custom logic, it is ultimately tied to the Firebase infrastructure, which can limit flexibility.

7.2 MSSQL Flexibility

MSSQL provides unparalleled flexibility.

References

1. RabbitMQ Quorum Queues: RabbitMQ official documentation provides insights into quorum queues, their durability, and how they implement the Raft Consensus Algorithm to ensure reliable message delivery. Quorum Queues Documentation
2. MSSQL Service Broker: Microsoft documentation explains how Service Broker allows applications to send and receive asynchronous messages, providing a mechanism for real-time communication. Microsoft Service Broker Documentation
3. ASP.NET Core SignalR: Microsoft's ASP.NET Core SignalR enables real-time web functionality, allowing server-side code to push content to connected clients instantly. ASP.NET Core SignalR Documentation
4. In-Memory OLTP: SQL Server In-Memory OLTP offers memory-optimized tables that improve performance for high-volume transactional workloads. SQL Server In-Memory OLTP Documentation
5. GPT-3 for AI Moderation: OpenAI's GPT-3 model has been adapted for text moderation tasks, helping identify harmful and abusive content in messaging platforms. OpenAI GPT-3 Model Overview
6. Firebase Real-Time Database: Firebase's NoSQL JSON data structure offers real-time data synchronization for applications but faces challenges with large-scale applications. Firebase Realtime Database Documentation

7. Change Data Capture (CDC) in MSSQL: Microsoft’s Change Data Capture feature allows real-time monitoring of database changes, facilitating real-time data-driven applications. CDC Documentation
8. SignalR WebSockets: SignalR supports WebSockets for maintaining real-time communication between clients and servers, ensuring low-latency message delivery. SignalR and WebSockets
9. AI-Based Post Moderation: AI-based models for text moderation, such as GPT, are becoming increasingly relevant in social platforms, providing scalability and efficiency in detecting inappropriate content. AI Text Moderation Overview
10. Cost Comparison Between Firebase and MSSQL: Cost comparisons between Firebase and MSSQL are critical when considering high-scale messaging systems. Firebase's costs can grow exponentially with usage, while MSSQL provides more predictable cost structures for large-scale operations. Firebase Pricing and SQL Server Pricing