DynamoDB Domain Models and Table Setup

Below are the **domain models** for the key entities in the system, along with their DynamoDB schema design. Each model outlines the primary keys and important attributes stored. After defining the models, sample **DynamoDB table creation scripts** (using AWS SDK in Python) are provided for each entity.

User Model

Represents an application user and stores profile and integration info.

- **UserId** (string, *Primary Key*): Unique identifier for the user (could be a UUID or username).
- **Name** (string): The user's name or display name.
- **Email** (string): Contact email of the user (could be used for login or notifications).
- **APIIntegrationDetails** (map/string): Credentials or settings for third-party AI integrations. For example, this may include an API key for an AI service or other config needed to connect on behalf of the user.
- **CreatedAt** (timestamp): Account creation date/time.
- *Additional attributes* like last login timestamp, account status, etc., can be included as needed.

DynamoDB Schema: The **Users** table uses `UserId` as the partition key. No sort key is needed since each user item is uniquely identified by `UserId`. (If queries by email or other attributes are required, secondary indexes could be added, but not shown here for brevity.)

Agent Model

Represents an AI agent owned by a user, including its configuration.

- **AgentId** (string, *Primary Key*): Unique identifier for the agent.

- **UserId** (string): Reference to the **User** who owns this agent. (This could be used as a partition key instead, if designing a composite key; here we store it as an attribute and can index it if needed for queries.)
- **Provider** (string): The AI backend provider or model this agent uses (e.g., `"OpenAI GPT-4"`, `"Azure AI"`). This indicates where the agent's AI logic is running.
- **Functions** (list/map): Definitions of any custom functions or tools the agent can use. This might be stored as a list of function metadata (names, descriptions, etc.) or a JSON structure.
- **SystemMessage** (string): A system or preset message for the agent (e.g., the initial prompt or persona definition that the agent always uses at the start of a conversation).
- **Settings** (map): Configuration settings for the agent's behavior. For example, this can include parameters like temperature, model version, max tokens, or toggles for certain features.

DynamoDB Schema: The **Agents** table uses `AgentId` as the partition key. Each item is one agent. The `UserId` is stored with each agent to know who owns it (optionally, a Global Secondary Index on `UserId` could be created to query all agents by a particular user).

Conversation Model

Represents a conversation session between a user and an agent.

- **ConversationId** (string, *Primary Key*): Unique identifier for the conversation session (could be a UUID).
- **UserId** (string): The user participating in the conversation.
- **AgentId** (string): The AI agent involved in the conversation.
- **StartedAt** (timestamp): Timestamp when the conversation began.
- **LastActiveAt** (timestamp): Timestamp of the latest activity in the conversation (e.g., last message sent). This can help in sorting or archiving old conversations.
- **Title** (string, optional): A human-friendly title or summary of the conversation (if the application allows naming conversations).

- **Status** (string, optional): Status of the conversation (e.g., `"active"`, `"completed"`, `"archived"`).

DynamoDB Schema: The **Conversations** table uses `ConversationId` as the partition key (unique per conversation). We also store `UserId` and `AgentId` as attributes to know the participants. (For retrieval, you might add a GSI on `UserId` to list all conversations for a user, or on `AgentId` to list all conversations handled by a specific agent. In a simple design, the application can also store conversation IDs in a list under the User or Agent item if needed, but here each conversation is a separate item.)

Message Model

Represents an individual message within a conversation.

- **ConversationId** (string, *Partition Key*): Identifier of the conversation this message belongs to (links to a Conversation item).
- **Timestamp** (number, *Sort Key*): Timestamp of when the message was sent. This serves as the sort key so that messages in the same conversation are ordered chronologically. (Using epoch time or a similar numeric timestamp ensures correct ordering. Alternatively, an ISO date string could be used as sort key if formatted to lexicographically sort by time.)
- **MessageId** (string, optional): A unique ID for the message (if needed, could also serve as sort key instead of timestamp, e.g., a ULID to ensure ordering). Often the combination of ConversationId and Timestamp is enough to uniquely identify a message.
- **Sender** (string): Who sent the message. For example `"User"` or `"Agent"` (or specific userId/agentId if needed for more detail). This could also be an enum like `"user"`, `"agent"`, or `"system"` (if system/agent are similar here).
- **Content** (string): The text content of the message. This is the main body of what was said by the sender.
- **Attributes** (map, optional): Any additional metadata for the message. For instance, this might include flags like `isFunctionCall: true` if the message represents a function call, or other message-specific data.

DynamoDB Schema: The **Messages** table is designed with a composite key: `ConversationId` (partition key) and `Timestamp` (sort key). This schema partitions messages by conversation, and within each conversation, messages are sorted by their timestamp. This design makes it efficient to query all messages for a conversation in order. The other fields (sender, content, etc.) are non-key attributes stored with each item.

DynamoDB Table Creation Scripts (Python Boto3)

Using the chosen language **Python**, we can create the DynamoDB tables for the above models with the AWS SDK (Boto3). Below is an example script that creates each table with the specified primary key schema. (For simplicity, these tables use on-demand capacity mode via `BillingMode='PAYPERREQUEST'` to avoid specifying read/write units.)

```
python
CopyEdit
import boto3

# Initialize DynamoDB client (or resource)
dynamodb = boto3.client('dynamodb')

# 1. Create Users table
dynamodb.create_table(
    TableName='Users',
    KeySchema=[{'AttributeName': 'UserId', 'KeyType': 'HASH'}],  # Partition ke
y: UserId
    AttributeDefinitions=[{'AttributeName': 'UserId', 'AttributeType': 'S'}],
    BillingMode='PAYPERREQUEST'
)

# 2. Create Agents table
dynamodb.create_table(
    TableName='Agents',
    KeySchema=[{'AttributeName': 'AgentId', 'KeyType': 'HASH'}],  # Partition k
ey: AgentId
```

```
    AttributeDefinitions=[{'AttributeName': 'AgentId', 'AttributeType': 'S'}],
    BillingMode='PAYPERREQUEST'
    # (Optionally, define a Global Secondary Index on UserId if needed for quer
ying by owner)
)

# 3. Create Conversations table
dynamodb.create_table(
    TableName='Conversations',
    KeySchema=[{'AttributeName': 'ConversationId', 'KeyType': 'HASH'}],  # Pa
rtition key: ConversationId
    AttributeDefinitions=[{'AttributeName': 'ConversationId', 'AttributeType':
'S'}],
    BillingMode='PAYPERREQUEST'
    # (Optionally, add indexes for UserId or AgentId to query by those attribute
s)
)

# 4. Create Messages table
dynamodb.create_table(
    TableName='Messages',
    KeySchema=[
        {'AttributeName': 'ConversationId', 'KeyType': 'HASH'},  # Partition key: C
onversationId
        {'AttributeName': 'Timestamp', 'KeyType': 'RANGE'}       # Sort key: Times
tamp
    ],
    AttributeDefinitions=[
        {'AttributeName': 'ConversationId', 'AttributeType': 'S'},
        {'AttributeName': 'Timestamp', 'AttributeType': 'N'}
    ],
    BillingMode='PAYPERREQUEST'
)
```

In this setup, each DynamoDB table is created with the primary keys corresponding to the domain model definitions:

- **Users** table keyed by `UserId` (string).
- **Agents** table keyed by `AgentId` (string).
- **Conversations** table keyed by `ConversationId` (string).
- **Messages** table keyed by composite of `ConversationId` (string) and `Timestamp` (number).

These tables and schemas will support storing and retrieving the data for users, AI agents, conversations, and messages in the application. Each table creation call can be further extended with secondary indexes or additional settings as needed by the application's query patterns.

You said:

is ddb best choice? should we just use postgres?

ChatGPT said:

To determine whether DynamoDB or PostgreSQL is the best choice for your use case, I need to clarify a few things:

- **Data Access Patterns:** How do you plan to query your data? Do you need complex queries, joins, and aggregations, or are simple key-value lookups sufficient?
- **Scalability Needs:** Is auto-scaling and high availability a priority, or are you fine with a single-node relational database?
- **Cost Considerations:** Are you optimizing for cost efficiency, or is performance more important?
- **Data Consistency Requirements:** Do you need strong ACID compliance (Postgres) or eventual consistency (DynamoDB is optimized for high availability but can support strong consistency in some cases)?
- **Schema Flexibility:** Do you need a rigid schema (PostgreSQL) or a flexible, document-like structure (DynamoDB)?

- **Existing Infrastructure:** Are you already using AWS services heavily, making DynamoDB a better fit, or do you have a preference for SQL and relational databases?

Let me know your priorities, and I can make a well-informed recommendation.

You said:

Low cost, scale isn't as important, simple queries mostly, schema could go either way, we would host in AWS either way.

ChatGPT said:

Given your priorities—low cost, simple queries, and AWS-hosted infrastructure—I will analyze whether DynamoDB or PostgreSQL is the best fit for your use case. I'll compare cost implications, operational overhead, query flexibility, and ease of integration within AWS. I'll get back to you with a clear recommendation soon.