# 1. Introduction

Amazon DynamoDB stands as a foundational service within the AWS ecosystem, providing a highly scalable and performant NoSQL database solution. Its architecture, designed for distributed operation, inherently involves asynchronous processes for certain management tasks. When interacting with DynamoDB, particularly during operations that alter the state or configuration of tables and their associated resources, a period of time is often required for these changes to propagate and become fully effective. Attempting to perform subsequent actions prematurely, before the underlying resources have reached a stable state, can lead to application errors, inconsistent behavior, and an overall degradation of the user experience. This report aims to provide a comprehensive overview of the key DynamoDB operations that necessitate waiting, elucidating the technical reasons behind these temporal requirements. Furthermore, it will offer guidance on effectively managing these wait times using the AWS Software Development Kit (SDK), thereby enabling developers to build more robust and dependable applications leveraging the power of DynamoDB.

# 2. Identifying DynamoDB Operations Requiring Wait Times

Several core DynamoDB operations involve asynchronous processes that necessitate a waiting period to ensure the reliability of subsequent interactions. These operations primarily revolve around modifying the structure or configuration of tables and their associated global secondary indexes (GSIs).

1. **CreateTable:** The creation of a new DynamoDB table initiates a series of resource allocation and initialization steps within the AWS infrastructure. During this phase, the table's status is typically reported as CREATING. Until this process is complete and the table status transitions to ACTIVE, operations such as putting, getting, updating, or deleting items cannot be reliably performed. Applications attempting to interact with a table in the CREATING state are likely to encounter errors as the underlying infrastructure is not yet fully provisioned to handle data operations.
2. **UpdateTable:** Modifying the properties of an existing DynamoDB table also requires a certain amount of time to take effect. These modifications can include adjustments to provisioned throughput (for tables in provisioned capacity mode), the addition or removal of global secondary indexes, or enabling/disabling features like point-in-time recovery or encryption. While these updates are in progress, the table's status may transition to UPDATING. Although some read and write operations might still be possible during specific phases of the update process, it is generally recommended to wait until the table returns to an ACTIVE state. This ensures that the requested changes have been successfully applied across the distributed system and that subsequent operations will reflect the updated configuration consistently.
3. **DeleteTable:** The process of deleting a DynamoDB table involves deallocating the associated resources and removing all data. When a DeleteTable request is initiated, the

table's status typically changes to DELETING. Once this deletion process commences, no further operations can be performed on the table. While applications do not typically perform actions on a table after initiating its deletion, it is important to recognize that the complete removal of the table is not an instantaneous event. The system requires time to decommission the resources and ensure the table no longer exists.

4. **UpdateGlobalSecondaryIndex:** Similar to updating a table, modifying a global secondary index, such as adjusting its provisioned throughput, requires time for the changes to propagate through the DynamoDB service. During this period, the GSI's status will likely be UPDATING. It is advisable to wait until the GSI's status becomes ACTIVE before relying on it for consistent and accurate query results. Querying an index that is still in the UPDATING state might yield inconsistent or incomplete data.
5. **DeleteGlobalSecondaryIndex:** The removal of a global secondary index also involves asynchronous processes. As with table deletion, the system needs to deallocate the resources associated with the index and remove its data. While this occurs, the GSI's status will be DELETING. Applications should be aware that this operation is not immediate, and the index will continue to exist in the DELETING state for a period before it is fully removed from the table's schema.

# 3. Deep Dive into the Reasons for Waiting

The necessity for waiting after initiating these DynamoDB operations arises from the fundamental characteristics of a distributed database system that prioritizes availability and durability. These operations are not instantaneous due to several underlying factors:

- **Resource Provisioning:** Creating or modifying DynamoDB tables and global secondary indexes involves the allocation and configuration of physical infrastructure resources within AWS. This includes storage, compute, and network resources that are distributed across multiple servers and availability zones. The process of provisioning these resources, ensuring they are correctly configured, and making them ready to handle traffic inherently takes a certain amount of time. The distributed nature of the system means that these resources need to be coordinated across different components, adding to the overall latency of the operation.
- **Data Replication and Indexing:** DynamoDB ensures high availability and data durability by replicating data across multiple availability zones within an AWS region.[1] When changes are made to a table's structure, such as adding or removing a global secondary index, the system needs to replicate the existing data to the new index structure across these multiple zones. Similarly, when a table or index is deleted, the data needs to be securely deallocated and removed from all replicas. These data replication and indexing processes are not immediate and contribute significantly to the wait times observed after operations like CreateTable, UpdateTable (especially when adding/removing GSIs), DeleteTable, UpdateGlobalSecondaryIndex, and DeleteGlobalSecondaryIndex.
- **Metadata Updates:** DynamoDB maintains a substantial amount of metadata about the state and configuration of all tables and indexes within an account. This metadata

includes information about the table schema, provisioned throughput, the status of GSIs, and various feature settings. When an operation that changes the state or configuration of a resource is initiated, DynamoDB needs to update this metadata. To ensure the consistent and correct operation of the entire system, these metadata updates must be propagated and applied across all relevant internal services and nodes. Achieving this consistency across a distributed system requires time and coordination, which is why a waiting period is necessary after state-changing operations.

# 4. The Significance of Waiting After CreateTable

Among the DynamoDB operations that require waiting, the period following the creation of a new table is particularly critical for application stability. If an application attempts to perform read or write operations (such as PutItem, GetItem, UpdateItem, or DeleteItem) immediately after sending a CreateTable request, it will likely encounter errors.[1] This is because the table is still in the CREATING state and is not yet prepared to serve traffic.

Attempting to interact with a non-active table can result in various exceptions, such as ResourceNotFoundException if the table metadata has not fully registered across the system, or general operation failures because the underlying storage and compute resources are not yet ready.[5] These errors can lead to unexpected application behavior, data loss, or a degraded user experience.

To avoid these issues, it is a fundamental best practice to ensure that a newly created DynamoDB table has reached the ACTIVE status before attempting any data plane operations on it.[1] The AWS SDK provides convenient mechanisms, such as the waitFor('tableExists',...) method in SDK v2 and the waitUntilTableExists function in SDK v3, to handle this asynchronous nature. These methods poll the DynamoDB service in the background, periodically checking the table's status until it becomes ACTIVE. By incorporating these waiting mechanisms into the application logic, developers can ensure that their code interacts with DynamoDB resources only when they are in a ready state, thereby preventing errors and ensuring the smooth operation of their applications.

# 5. Leveraging the AWS SDK for Handling Wait Times

The AWS SDK for JavaScript provides effective tools for managing the wait times associated with asynchronous DynamoDB operations, ensuring that applications interact with resources only when they are in the desired state.

## 5.1 Introduction to waitFor (v2) and waitUntilTableExists (v3)

Both versions of the AWS SDK for JavaScript offer functionalities to abstract away the complexity of manually polling DynamoDB for resource status. In version 2 of the SDK, the AWS.DynamoDB client class includes a waitFor method.[6] This method allows developers to wait for a specific resource state to be reached before proceeding with further actions. For DynamoDB tables, the waitFor method supports two primary states: 'tableExists', which waits

until the specified table exists and its status is ACTIVE, and 'tableNotExists', which waits until the table is deleted and no longer exists.[9]
In contrast, version 3 of the AWS SDK for JavaScript adopts a modular approach, where functionalities are available as separate functions. For waiting for a DynamoDB table to exist and become active, SDK v3 provides the waitUntilTableExists function from the @aws-sdk/client-dynamodb package.[2] This function serves a similar purpose to the waitFor('tableExists',...) method in v2, polling the DynamoDB service until the specified table's status is ACTIVE. While the provided research material primarily focuses on waiting for table existence, it's worth noting that v3 also offers other waitUntil functions for different DynamoDB resource states, although these were not explicitly detailed in the snippets. The key difference lies in the architecture: waitFor is bundled within the v2 client, while waitUntil functions are imported separately in v3, promoting a smaller bundle size for applications that do not require waiting functionality.[12]

## 5.2 Detailed Explanation of the TypeScript Code Example

The provided TypeScript code demonstrates the use of the waitFor method in AWS SDK version 2.159 to create a DynamoDB table and wait for it to become active before performing a subsequent operation.
The code begins by importing the necessary AWS SDK module.[2] It then configures the AWS SDK with a specified region (ap-south-1 in this example), which should be replaced with the desired AWS region. An instance of the AWS.DynamoDB client is created, which will be used to interact with the DynamoDB service. The code defines a tableName variable and constructs a params object that specifies the properties of the DynamoDB table to be created. These parameters include the TableName, the KeySchema defining the primary key attributes (id as a hash key in this case), the AttributeDefinitions specifying the data type of the key attribute, and the ProvisionedThroughput settings for read and write capacity units. It is crucial to replace the placeholder values for the region, table name, key schema, attribute definitions, and provisioned throughput with the actual requirements for the table being created.
The asynchronous function createAndWaitForTable encapsulates the table creation and waiting logic. It first logs a message indicating that the table creation process has been initiated. The dynamodb.createTable(params).promise() method sends the request to create the table. The .promise() converts the callback-based function to a Promise, enabling the use of async/await for cleaner asynchronous code. Upon successful initiation of table creation, the code logs the table's Amazon Resource Name (ARN) if available in the response.
The core of the waiting mechanism is the line await dynamodb.waitFor('tableExists', { TableName: tableName }).promise();. The dynamodb.waitFor('tableExists',...) method is invoked to poll the DynamoDB service. The 'tableExists' argument specifies that the waiter should continue polling until the table's status becomes ACTIVE. The { TableName: tableName } object provides the necessary parameter for the underlying DescribeTable API call that waitFor uses internally to check the table's status. The .promise() again converts the callback-based waitFor function to a Promise, ensuring that the code execution pauses at this point until the table is active or an error occurs. Once the waitFor promise resolves, a

message is logged indicating that the table is now active. As an example of a subsequent operation that can be performed safely, the code then calls dynamodb.describeTable({ TableName: tableName }).promise() to retrieve and log the table's description, specifically its TableStatus.
The entire process is wrapped in a try...catch block to handle any potential errors that might occur during table creation or while waiting for the table to become active. If an error is caught, an error message is logged to the console. Finally, the createAndWaitForTable() function is called to execute the table creation and waiting process. The comments in the code provide clear instructions on how to use this script, including the necessary installation steps for the AWS SDK and how to run the TypeScript file after compilation.

TypeScript

```
import AWS from 'aws-sdk';

// Configure AWS (replace with your actual configuration)
AWS.config.update({
  region: 'ap-south-1', // Replace with your desired region
  accessKeyId: 'YOUR_ACCESS_KEY_ID', // Replace with your access key
  secretAccessKey: 'YOUR_SECRET_ACCESS_KEY' // Replace with your secret key
});

const dynamodb = new AWS.DynamoDB({ apiVersion: '2012-08-10' });
const tableName = 'MyNewTable'; // Replace with your desired table name

const params: AWS.DynamoDB.CreateTableInput = {
  TableName: tableName,
  KeySchema:,
  AttributeDefinitions:,
  ProvisionedThroughput: {
    ReadCapacityUnits: 5, // Adjust as needed
    WriteCapacityUnits: 5 // Adjust as needed
  }
};

async function createAndWaitForTable() {
  try {
    console.log(`Creating table ${tableName}...`);
    const createTableResponse = await dynamodb.createTable(params).promise();
    console.log('Table created:', createTableResponse.TableDescription?.TableArn);

    console.log(`Waiting for table ${tableName} to become active...`);
```

```
    await dynamodb.waitFor('tableExists', { TableName: tableName }).promise();
    console.log(`Table ${tableName} is now active.`);

    // Example of performing an operation after the table is active
    const describeTableResponse = await dynamodb.describeTable({ TableName: tableName
}).promise();
    console.log('Table description:', describeTableResponse.Table?.TableStatus);

  } catch (error) {
    console.error('Error creating or waiting for table:', error);
  }
}

createAndWaitForTable();

/*
To run this code:
1. Ensure you have Node.js and npm installed.
2. Install the AWS SDK for JavaScript (v2): npm install aws-sdk
3. Save this code as a TypeScript file (e.g., create-table.ts).
4. Compile the TypeScript file: npx tsc create-table.ts
5. Run the JavaScript file: node create-table.js

Remember to replace the placeholder AWS credentials and region with your actual values.
*/
```

## 5.3 Configuration Options for Waiters (Timeout and Retry)

The waitFor method in AWS SDK v2 and the waitUntilTableExists function in SDK v3 have implicit and explicit configuration options that control their behavior, particularly regarding timeouts and retries. In SDK v2, the waitFor method for DynamoDB's 'tableExists' and 'tableNotExists' states has a default polling interval of 20 seconds and a maximum of 25 attempts.[8] This means that, by default, the waiter will continue to check the table's status for up to approximately 500 seconds (around 8.3 minutes) before timing out. The delay between each poll is fixed at 20 seconds, representing a linear backoff strategy.[12] While there isn't a direct parameter to set a total timeout for waitFor in v2, the maxAttempts indirectly serves this purpose. It is possible to customize the delay between polls by using the $waiter parameter within the options object passed to waitFor, with the delay option specifying the wait time in seconds.[12] For example, to set a delay of 5 seconds and a maximum of 10 attempts:

JavaScript

```
dynamodb.waitFor('tableExists', { TableName: 'YourTableName', $waiter: { delay: 5,
maxAttempts: 10 } }, (err, data) => {
  if (err) {
    console.error('Error waiting:', err);
  } else {
    console.log('Table exists:', data);
  }
});
```

In SDK v3, the waitUntilTableExists function offers more explicit control over the waiting process. It accepts a configuration object as its first parameter, which can include options like maxWaitTime, specifying the maximum number of seconds to wait for the table to reach the ACTIVE state.[12] This provides a direct way to set a timeout for the waiting operation. Additionally, v3 employs an exponential backoff strategy with full jitter for retries, which generally leads to more efficient polling compared to v2's linear backoff.[12] This strategy introduces randomness into the delay between retries, helping to prevent thundering herd issues and distribute the load on the service more effectively. For instance, to set a maximum wait time of 60 seconds, a minimum delay of 2 seconds, and a maximum delay of 5 seconds:

JavaScript

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { waitUntilTableExists } from "@aws-sdk/client-dynamodb";

const client = new DynamoDBClient({ region: 'your-region' });
const tableName = 'YourTableName';

async function waitForActive() {
  try {
    await waitUntilTableExists({ client, maxWaitTime: 60, minDelay: 2, maxDelay: 5 }, {
TableName: tableName });
    console.log(`Table ${tableName} is now active.`);
  } catch (error) {
    console.error("Error waiting for table to exist:", error);
  }
}

waitForActive();
```

It is also important to note that the AWS SDK, in general, allows for configuring timeouts at the client level. For instance, in both v2 and v3, options like httpOptions.timeout and

httpOptions.connectTimeout (v2) or socketTimeout and connectionTimeout within the NodeHttpHandler configuration (v3) can be set during client initialization.[19] These settings define the timeout for individual HTTP requests made by the SDK, including those made by the waiter functions. Configuring these client-level timeouts can provide an additional layer of control over the overall duration of the waiting process. The default timeout for v2's waitFor (500 seconds) might be excessive for some applications, highlighting the importance of considering and potentially adjusting these settings based on the expected operation duration and the application's tolerance for latency. SDK v3's maxWaitTime offers a more straightforward approach to managing the total waiting period.

### 5.4 Error Handling with Waiters

Proper error handling is crucial when using the waitFor method in SDK v2 or the waitUntilTableExists function in SDK v3. As demonstrated in the provided TypeScript code example, these asynchronous operations should be wrapped in try...catch blocks to manage potential errors.[22] One of the primary errors to anticipate is a timeout. If the waitFor or waitUntil function exceeds its configured timeout period (either the default or a custom value set by the developer) before the resource reaches the desired state, it will typically reject the promise with an error.[16] This error will often indicate that the maximum wait time was reached without the table (or index) becoming active or not existing, as the case may be.

When such a timeout error occurs, it is important for the application to log the error details, including the table name and the specific state being waited for. Depending on the application's requirements and the nature of the operation, it might be appropriate to implement retry logic at a higher level. For example, if table creation fails to become active within the initial timeout, the application might attempt to recreate the table after a certain delay, again incorporating a waiting mechanism. It's also possible that the operation genuinely failed due to an underlying issue with the DynamoDB service or the provided parameters. In such cases, retrying indefinitely might not be the best approach, and the application should consider alternative actions, such as notifying an administrator or failing gracefully.

General error handling best practices for interacting with DynamoDB, as outlined in the research material, also apply when using waiters.[22] This includes inspecting the error object for details such as the HTTP status code, exception name, and error message, which can provide valuable insights into the cause of the failure. For instance, a ResourceNotFoundException during a waitFor('tableExists',...) call might indicate that the table name was incorrect. By implementing comprehensive error handling around the use of waiter functions, developers can build more resilient applications that can gracefully handle the asynchronous nature of DynamoDB operations and recover from potential failures.

## 6. General Best Practices for Managing Asynchronous DynamoDB Operations

Effectively managing asynchronous DynamoDB operations is essential for building reliable and performant applications. Here are several best practices to consider:

- **Always Use Waiters for State-Changing Operations:** For operations that modify the state or configuration of DynamoDB tables and global secondary indexes, such as CreateTable, UpdateTable, DeleteTable, UpdateGlobalSecondaryIndex, and DeleteGlobalSecondaryIndex, it is crucial to use the waitFor (v2) or waitUntil (v3) methods provided by the AWS SDK.[1] This ensures that subsequent operations that depend on the resource being in a specific state are performed only after that state has been reached, preventing errors and inconsistencies.
- **Configure Appropriate Timeouts:** Relying solely on the default timeout settings for waiter functions might not be suitable for all applications. It is important to assess the expected duration of each state-changing operation and configure timeout values accordingly.[19] Setting reasonable timeouts helps prevent applications from waiting indefinitely for an operation that might be taking longer than expected or might have failed. Consider the application's latency requirements and set timeouts that strike a balance between allowing enough time for the operation to complete and failing fast if an issue occurs.
- **Implement Error Handling:** Asynchronous operations, including the waiter functions, can fail due to various reasons, such as network issues, service outages, or incorrect configurations. It is therefore essential to always include try...catch blocks around these operations to handle potential errors, including timeouts.[22] Logging error details can be invaluable for diagnosing and resolving issues.
- **Consider Retry Logic:** For critical asynchronous operations, implementing retry mechanisms at a higher level in the application can enhance resilience.[23] If a waiter function times out or fails due to a transient issue, retrying the entire sequence (initiating the operation and then waiting) with an appropriate backoff strategy (e.g., exponential backoff) can improve the chances of success on subsequent attempts.
- **Monitor Table and Index Status:** Actively monitoring the status of DynamoDB tables and global secondary indexes, especially during and after creation or update operations, can provide valuable insights into the health and progress of these resources. The AWS Management Console and the AWS CLI offer tools to check the current status of tables and indexes.
- **Understand Eventual Consistency:** While waiter functions ensure that a GSI reaches an ACTIVE state, it's important to remember DynamoDB's eventual consistency model, particularly concerning GSIs.[24] There might be a short delay between when a write operation is performed on the base table and when that change is reflected in the GSI. Applications that require strongly consistent reads should be aware of this and might need to adjust their design or consider using local secondary indexes where strongly consistent reads are supported.
- **Initialize Clients Outside of Request Handlers:** In serverless environments like AWS Lambda, initializing the DynamoDB client outside of the main request handler function allows for connection reuse across multiple invocations, which can significantly reduce latency and improve performance, especially when performing multiple asynchronous operations.[27]

- **Use Batch Operations:** When dealing with multiple independent read or write operations, utilizing batch operations like BatchGetItem and BatchWriteItem can be more efficient than performing individual operations, as they reduce the overhead of making multiple API calls.[23] However, it is important to handle potential errors and unprocessed items that might be returned in the response.
- **Design for Asynchronous Behavior:** Architecting applications to inherently handle the asynchronous nature of DynamoDB operations is a key best practice. This might involve using event-driven patterns, where an operation on DynamoDB triggers subsequent actions through mechanisms like DynamoDB Streams, rather than relying on synchronous, blocking calls.

# 7. Conclusion

In conclusion, understanding and correctly managing the wait times associated with asynchronous DynamoDB operations is paramount for building robust and reliable applications. Operations such as creating, updating, and deleting tables and global secondary indexes require a period of time to complete due to the underlying processes of resource provisioning, data replication, and metadata updates in this distributed database service. Attempting to interact with resources that are still in a transitional state can lead to various errors and unexpected application behavior.

The AWS SDK for JavaScript provides essential tools in the form of the waitFor method (in v2) and waitUntil functions (in v3) that simplify the process of handling these asynchronous operations. By using these methods, developers can ensure that their code interacts with DynamoDB resources only when they have reached the desired state. Furthermore, configuring appropriate timeouts, implementing robust error handling, and adhering to general best practices for managing asynchronous operations are crucial for optimizing application performance and resilience. By paying close attention to these considerations, developers can effectively leverage the scalability and performance of AWS DynamoDB to build stable and efficient applications that provide a positive user experience.

**Table 1: DynamoDB Operations Requiring Wait Times**

| Operation Name | Description | Typical Status During Operation | Recommended Wait Condition | Relevant AWS SDK Method(s) |
|---|---|---|---|---|
| CreateTable | Creates a new DynamoDB table. | CREATING | Status becomes ACTIVE | waitFor('tableExists'), waitUntilTableExists |
| UpdateTable | Modifies properties of an existing table. | UPDATING | Status becomes ACTIVE | waitFor('tableExists'), waitUntilTableExists |
| DeleteTable | Deletes a DynamoDB table and all of its items. | DELETING | Table does not exist | waitFor('tableNotExists') |

| | | | | |
|---|---|---|---|---|
| UpdateGlobalSecondaryIndex | Modifies properties of an existing global secondary index. | UPDATING | Status becomes ACTIVE | (Potentially custom polling) |
| DeleteGlobalSecondaryIndex | Deletes an existing global secondary index from a table. | DELETING | Index does not exist | (Potentially custom polling) |

**Table 2: AWS SDK Waiter Configuration**

| SDK Version | Wait Function | Timeout Configuration | Retry Configuration |
|---|---|---|---|
| v2 | waitFor | Implicit: ~500 seconds (25 attempts * 20 seconds) | Linear backoff: Fixed 20-second delay between attempts |
| v2 | waitFor | Explicit: $waiter: { delay: seconds } | Linear backoff: Custom delay |
| v3 | waitUntilTableExists | Explicit: maxWaitTime: seconds | Exponential backoff with full jitter (configurable min/max) |
| v3 | waitUntilTableExists | Implicit: Service-defined defaults (can be long) | Exponential backoff with full jitter (service defaults) |

## Works cited


1. Creating and Configuring DynamoDB Tables with ... - CodeSignal, accessed April 4, 2025, https://codesignal.com/learn/courses/introduction-to-dynamodb-with-aws-sdk-for-python/lessons/creating-and-configuring-dynamodb-tables-with-aws-sdk-for-python
2. DynamoDB examples using SDK for JavaScript (v3) - AWS Documentation, accessed April 4, 2025, https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/javascript_dynamodb_code_examples.html
3. Use CreateTable with an AWS SDK or CLI - Amazon DynamoDB, accessed April 4, 2025, https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/example_dynamodb_CreateTable_section.html
4. How To Remove DynamoDB Local And Test With AWS Managed DynamoDB - bahr.dev, accessed April 4, 2025, https://bahr.dev/2021/04/09/replacing-local-with-managed-dynamodb/
5. [Solved] DynamoDB cannot do operations on a non-existent table, accessed April

4, 2025, https://dynobase.dev/dynamodb-errors/dynamodb-cannot-do-operations-on-a-non-existent-table/
6. Class: AWS.DynamoDB — AWS SDK for JavaScript - AWS Documentation, accessed April 4, 2025, https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html
7. How to check DynamoDB Status while changing Throughput of WriteUnits and wait for it to complete? - Stack Overflow, accessed April 4, 2025, https://stackoverflow.com/questions/29251979/how-to-check-dynamodb-status-while-changing-throughput-of-writeunits-and-wait-fo
8. DynamoDB waitFor() with custom interval/max_attempts · Issue #881 · aws/aws-sdk-js, accessed April 4, 2025, https://github.com/aws/aws-sdk-js/issues/881
9. Class: AWS.DynamoDB — AWS SDK for JavaScript, accessed April 9, 2025, https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/DynamoDB.html#waitFor-property
10. waitUntilTableExists Variable - AWS SDK for JavaScript v3, accessed April 4, 2025, https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/Package/-aws-sdk-client-dynamodb/Variable/waitUntilTableExists
11. Class: AWS.BedrockRuntime — AWS SDK for JavaScript - AWS Documentation, accessed April 4, 2025, https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/BedrockRuntime.html
12. Waiters in modular AWS SDK for JavaScript | AWS Developer Tools Blog, accessed April 4, 2025, https://aws.amazon.com/blogs/developer/waiters-in-modular-aws-sdk-for-javascript/
13. @aws-sdk/client-dynamodb - npm, accessed April 4, 2025, https://www.npmjs.com/package/@aws-sdk/client-dynamodb
14. aws-sdk/client-dynamodb, accessed April 4, 2025, https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/Package/-aws-sdk-client-dynamodb
15. Auto Scaling examples using SDK for JavaScript (v3) - AWS Documentation - Amazon.com, accessed April 4, 2025, https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/javascript_auto-scaling_code_examples.html
16. Waiter doesn't error on Timeout · Issue #1917 · aws/aws-sdk-js-v3 - GitHub, accessed April 4, 2025, https://github.com/aws/aws-sdk-js-v3/issues/1917
17. accessed January 1, 1970, https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/client-dynamodb/function/waituntiltableexists/
18. accessed January 1, 1970, https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/client-dynamodb/classes/dynamodbclient/
19. Retry strategy in the AWS SDK for JavaScript v2, accessed April 4, 2025,

https://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/retry-strategy.html
20. AWS SDK Timeouts for JavaScript - jQN - Medium, accessed April 4, 2025, https://jqn.medium.com/aws-sdk-timeouts-for-javascript-967b86b100ee
21. amazon dynamodb - How do I set a timeout for AWS V3 Dynamo Clients - Stack Overflow, accessed April 4, 2025, https://stackoverflow.com/questions/66535671/how-do-i-set-a-timeout-for-aws-v3-dynamo-clients
22. Error handling with DynamoDB - Amazon DynamoDB, accessed April 4, 2025, https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Programming.Errors.html
23. Best Practices for ETL with S3, Lambda, and DynamoDB: Detailed Guide with Code Examples | by Mahtab Haider | Medium, accessed April 9, 2025, https://medium.com/@haider.mtech2011/best-practices-for-etl-with-s3-lambda-and-dynamodb-detailed-guide-with-code-examples-13a121cb49dc
24. Understanding Global Secondary Indexes in Amazon DynamoDB with the Rust SDK, accessed April 9, 2025, https://www.youtube.com/watch?v=O3dZU--wWRo
25. DynamoDB Secondary Indexes Archives - Jayendra's Cloud Certification Blog, accessed April 9, 2025, https://jayendrapatil.com/tag/dynamodb-secondary-indexes/
26. DynamoDB consistent reads for Global Secondary Index - Stack Overflow, accessed April 9, 2025, https://stackoverflow.com/questions/35414372/dynamodb-consistent-reads-for-global-secondary-index
27. Why is my dynamoDB queries taking so much time? - Stack Overflow, accessed April 9, 2025, https://stackoverflow.com/questions/76961827/why-is-my-dynamodb-queries-taking-so-much-time
28. DynamoDBClient - AWS SDK for JavaScript v3, accessed April 9, 2025, https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/client/dynamodb
29. 40 DynamoDB Best Practices [Bite-sized Tips] - Dynobase, accessed April 9, 2025, https://dynobase.dev/dynamodb-best-practices/