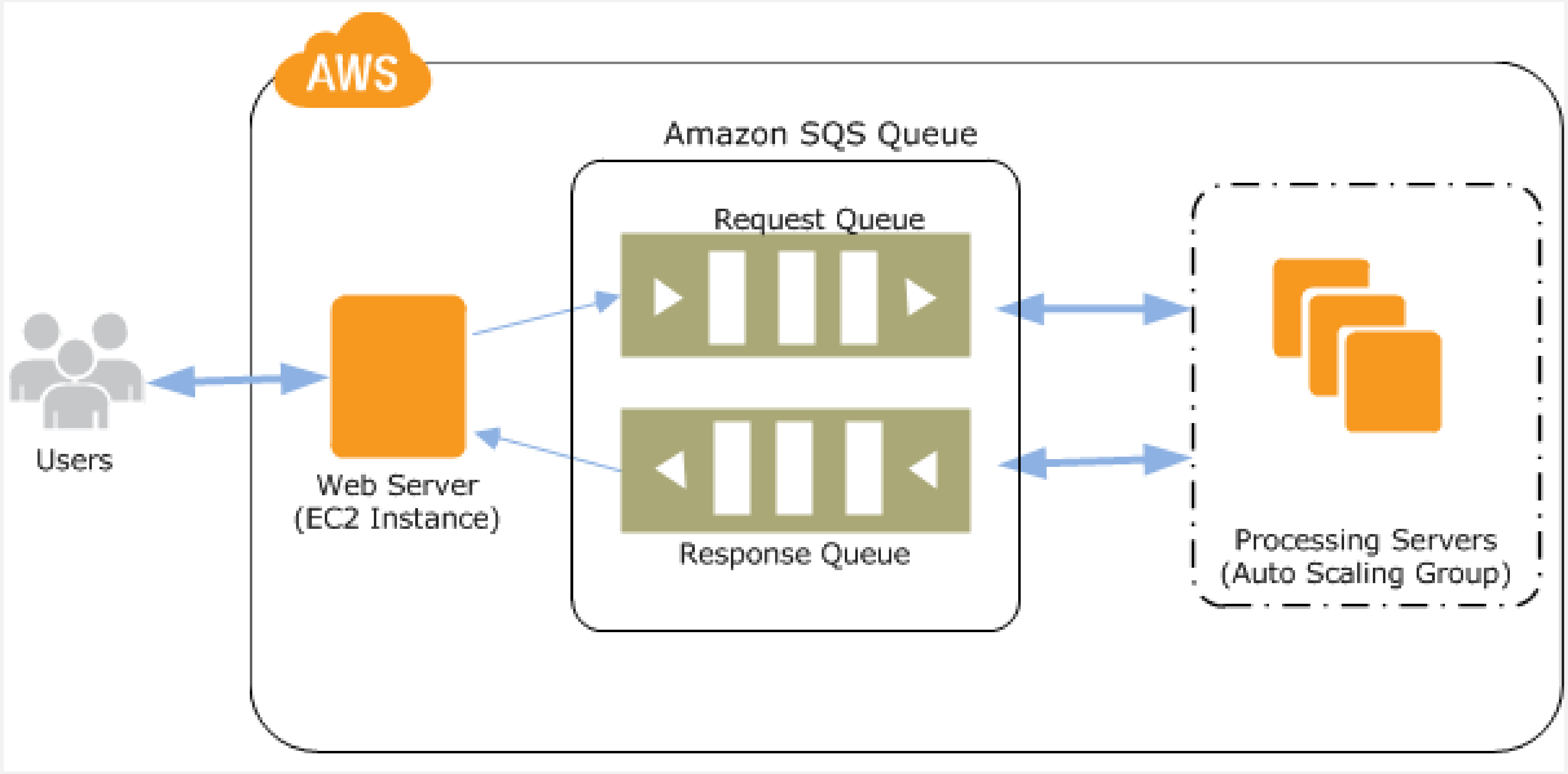
SAAS FOR QUEUING, NOTIFYING & CACHING

AMAZON SIMPLE QUEUE SERVICE

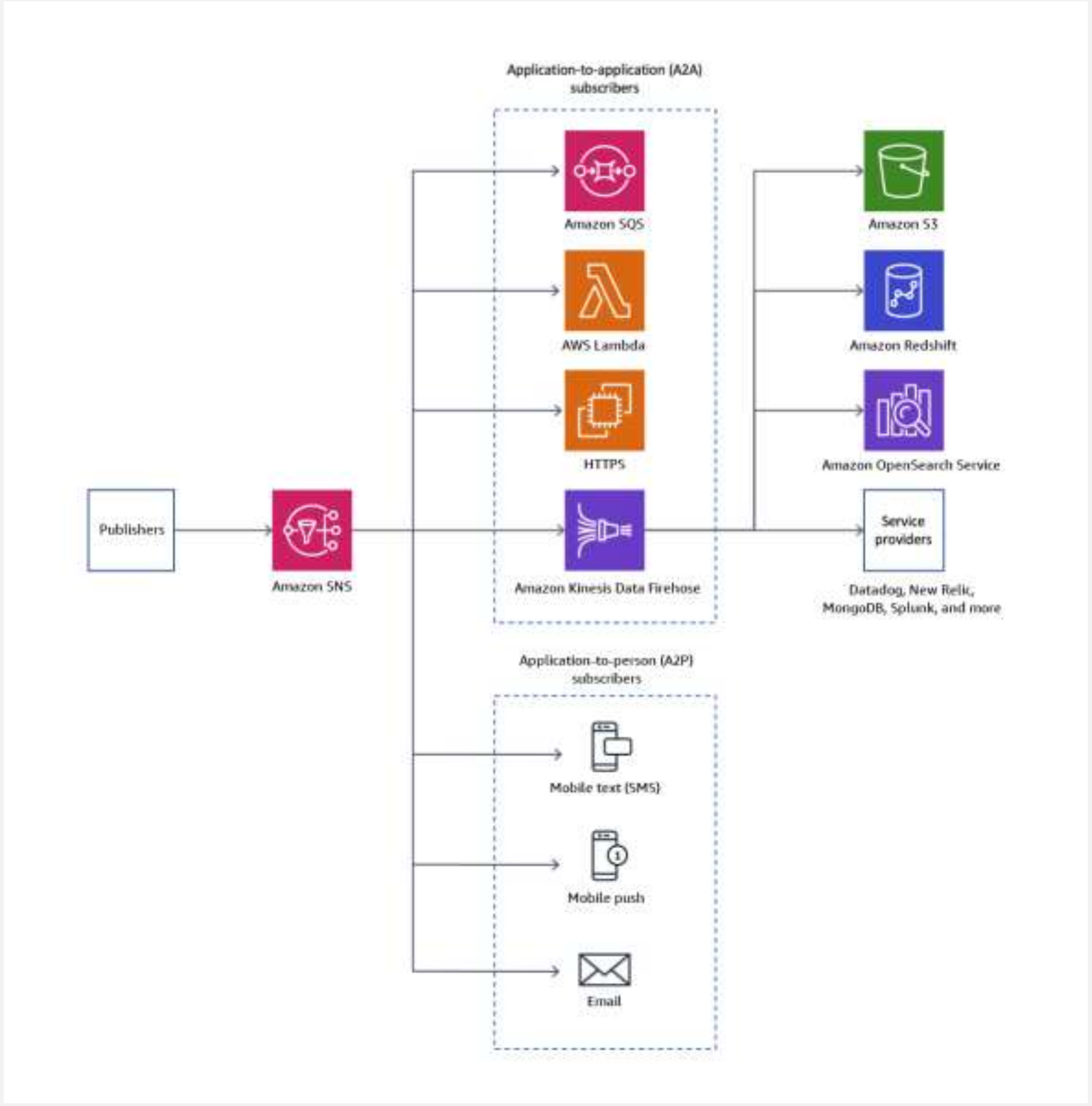


- Amazon Simple Queue Service (Amazon SQS) offers a secure, durable, and available hosted queue that lets you integrate and decouple distributed software systems and components.

- Amazon SQS offers common constructs such as dead-letter queues and cost allocation tags. It provides a generic web services API that you can access using any programming language that the AWS SDK supports.

- Amazon SQS supports both standard and FIFO queues.

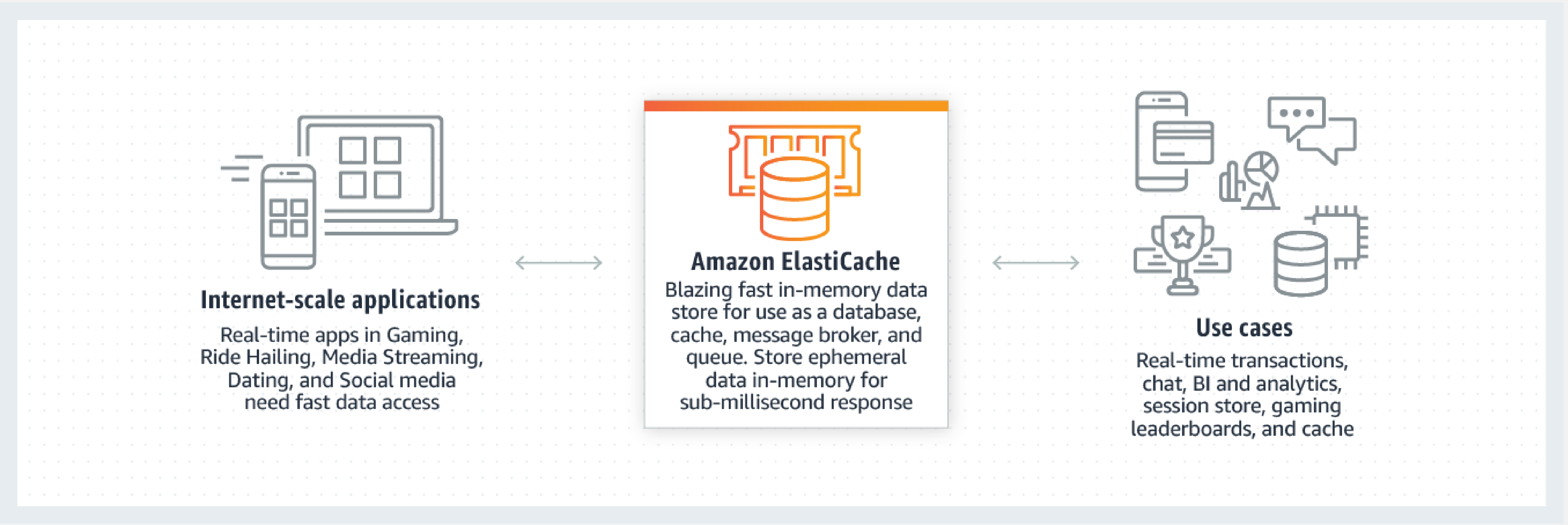
AMAZON SIMPLE NOTIFICATION SERVICE


Application-to-application (A2A) subscribers
Amazon SQS
AWS Lambda
HTTPS
Amazon Kinesis Data Firehose
Amazon S3
Amazon Redshift
Amazon OpenSearch Service
Service providers
Datadog, New Relic, MongoDB, Splunk, and more
Publishers
Amazon SNS
Application-to-person (A2P) subscribers
Mobile text (SMS)
Mobile push
Email

- Amazon Simple Notification Service (Amazon SNS) is a managed service that provides message delivery from publishers to subscribers (also known as *producers* and *consumers*).

- Publishers communicate asynchronously with subscribers by sending messages to a *topic*, which is a logical access point and communication channel. Clients can subscribe to the SNS topic and receive published messages using a supported endpoint type, such as Amazon Kinesis Data Firehose, Amazon SQS, AWS Lambda, HTTP, email, mobile push notifications, and mobile text messages (SMS).

AMAZON ELASTICACHE



Internet-scale applications

Real-time apps in Gaming, Ride Hailing, Media Streaming, Dating, and Social media need fast data access

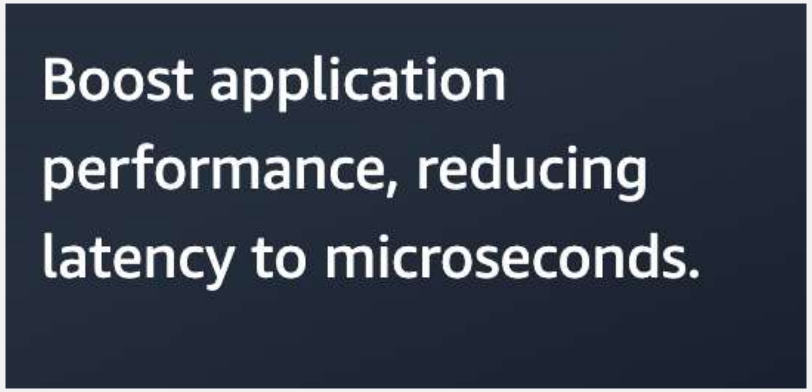
Amazon ElastiCache

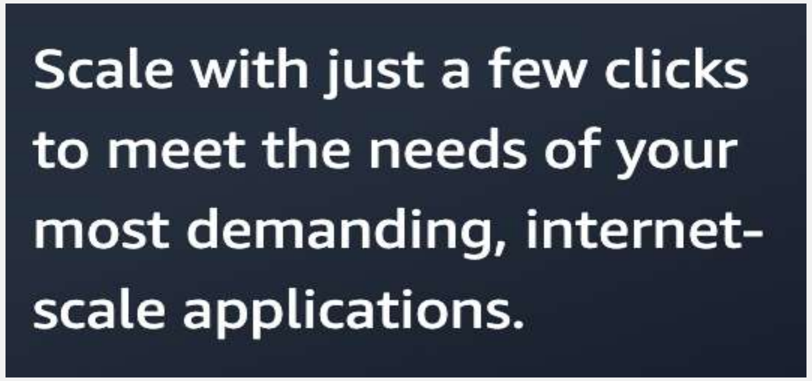
Blazing fast in-memory data store for use as a database, cache, message broker, and queue. Store ephemeral data in-memory for sub-millisecond response

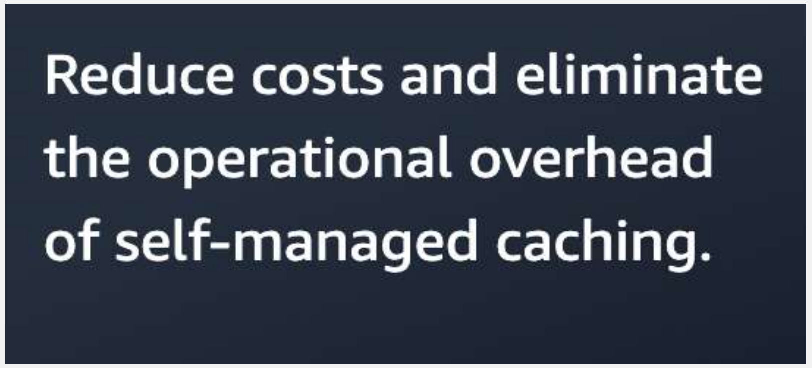
Use cases

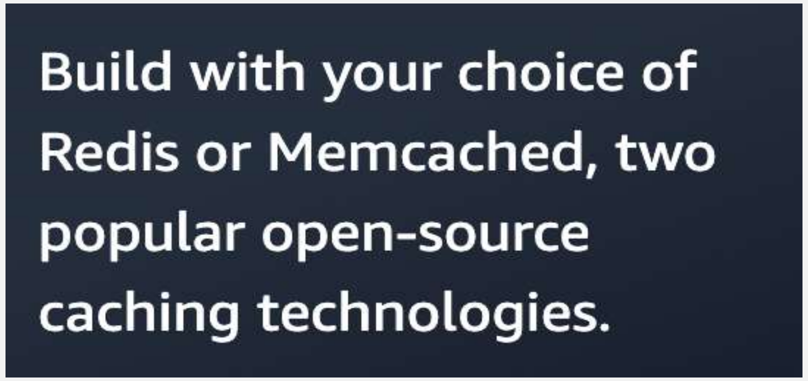
Real-time transactions, chat, BI and analytics, session store, gaming leaderboards, and cache

- Amazon ElastiCache is a fully managed, in-memory caching service supporting flexible, real-time use cases. You can use ElastiCache for caching, which accelerates application and database performance, or as a primary data store for use cases that don't require durability like session stores, gaming leaderboards, streaming, and analytics. ElastiCache is compatible with Redis and Memcached.


Boost application performance, reducing latency to microseconds.


Scale with just a few clicks to meet the needs of your most demanding, internet-scale applications.

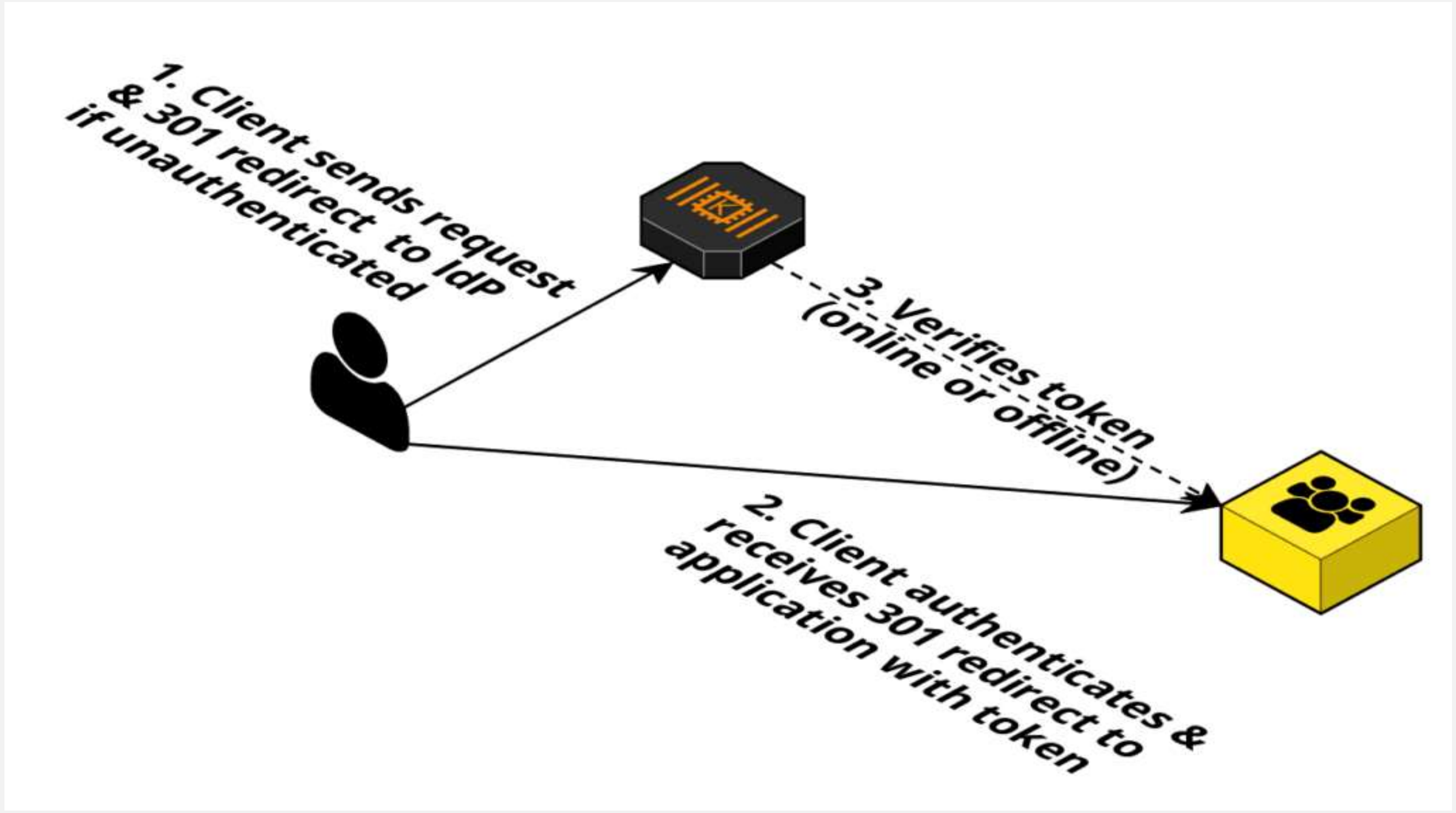

Reduce costs and eliminate the operational overhead of self-managed caching.


Build with your choice of Redis or Memcached, two popular open-source caching technologies.

CACHING STRATEGIES

- **Lazy loading-** As the name implies, *lazy loading* is a caching strategy that loads data into the cache only when necessary.
- **Write-through-** The write-through strategy adds data or updates data in the cache whenever data is written to the database.
- **Adding TTL-** *Time to live (TTL)* is an integer value that specifies the number of seconds until the key expires. Memcached specifies this value in seconds. When an application attempts to read an expired key, it is treated as though the key is not found. The database is queried for the key and the cache is updated. This approach doesn't guarantee that a value isn't stale. However, it keeps data from getting too stale and requires that values in the cache are occasionally refreshed from the database.

AUTHENTICATING YOUR APPLICATIONS TO AWS

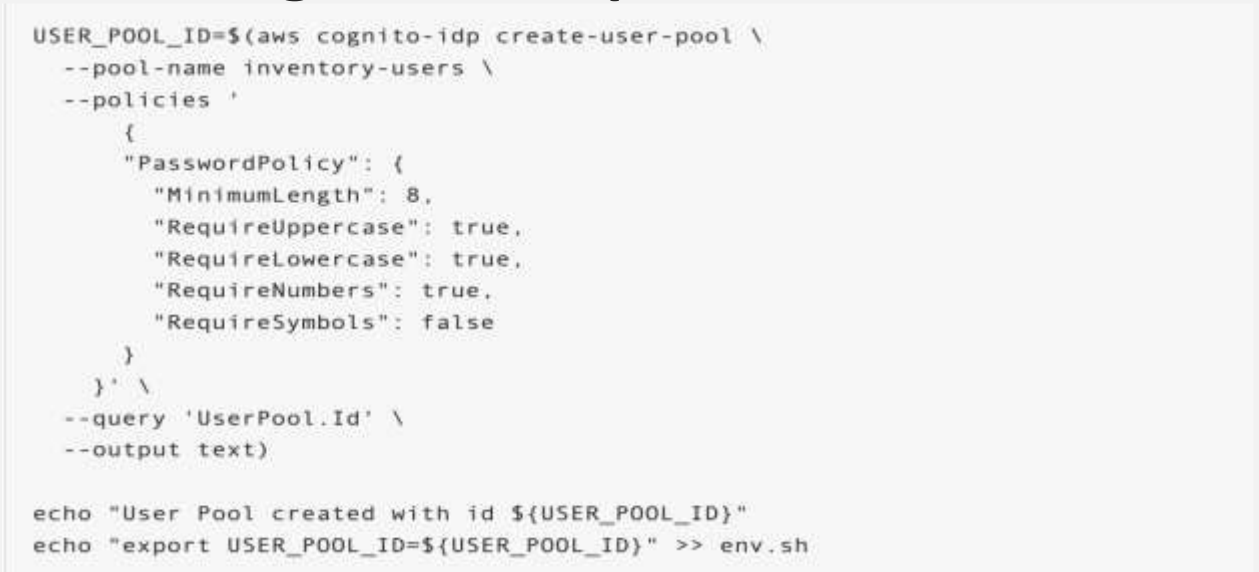

1. Client sends request & 301 redirect to IdP if unauthenticated
2. Client authenticates & receives 301 redirect to application with token
3. Verifies token (online or offline)

we will see how we can use AWS Cognito for Authentication & Authorization for a Web App in a completely serverless way. We will walk through the steps for enabling Signup, Sign in and Authorization to access to an AWS Resource.

- Configure a **Cognito User Pool** for a *User Authentication* process.
- Configure a **Cognito Identity Pool** (Federated Identities) for *Authorizing* the users to AWS resources.
- Configure **IAM Policy** for what AWS Resources users can access.
- Configure the specific AWS resource, in our case, it will be AWS S3.
- Start using the AWS Cognito Authentication/Authorization in your Web App.

BUILDING AN END-TO-END APP WITH AUTHENTICATION

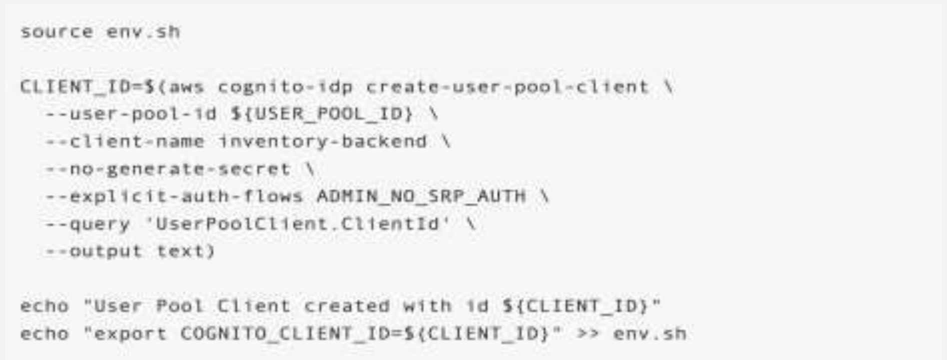
- Configure Amazon Cognito for use as the authentication provider in your application. Amazon Cognito is a fully managed authentication provider that allows for user sign-up, verification, login, and more.
- We create an Amazon Cognito user pool. Then you create a client to access the user pool.
- **Step 1. Create an Amazon Cognito user pool**



```
USER_POOL_ID=$(aws cognito-idp create-user-pool \
  --pool-name inventory-users \
  --policies '
      {
      "PasswordPolicy": {
        "MinimumLength": 8,
        "RequireUppercase": true,
        "RequireLowercase": true,
        "RequireNumbers": true,
        "RequireSymbols": false
      }
    }' \
  --query 'UserPool.Id' \
  --output text)

echo "User Pool created with id ${USER_POOL_ID}"
echo "export USER_POOL_ID=${USER_POOL_ID}" >> env.sh
```

- This script uses the AWS Command Line Interface (AWS CLI) to create a user pool. You give your user pool a name -- *inventory-users* -- and specify your password policies.
- create your user pool by executing the script with the following command:
- **$ bash scripts/create-user-pool.sh**
- You should see the following output:
- **User Pool created with id <user-pool-id>**
- **Step 2. Create a user pool client**
- There is a file in the scripts/ directory called create-user-pool-client.sh for creating a user pool client. The contents of the file are as follows:



```
source env.sh

CLIENT_ID=$(aws cognito-idp create-user-pool-client \
  --user-pool-id ${USER_POOL_ID} \
  --client-name inventory-backend \
  --no-generate-secret \
  --explicit-auth-flows ADMIN_NO_SRP_AUTH \
  --query 'UserPoolClient.ClientId' \
  --output text)

echo "User Pool Client created with id ${CLIENT_ID}"
echo "export COGNITO_CLIENT_ID=${CLIENT_ID}" >> env.sh
```

- Run the script to create the user pool client with the following command:
- **$ bash scripts/create-user-pool-client.sh**
- You should see the following output:
- **User Pool Client created with id <client-id>**
- **Step 3. Review authentication code**
- you have created a user pool and a client to access the pool, let’s see how you use Amazon Cognito in your application.

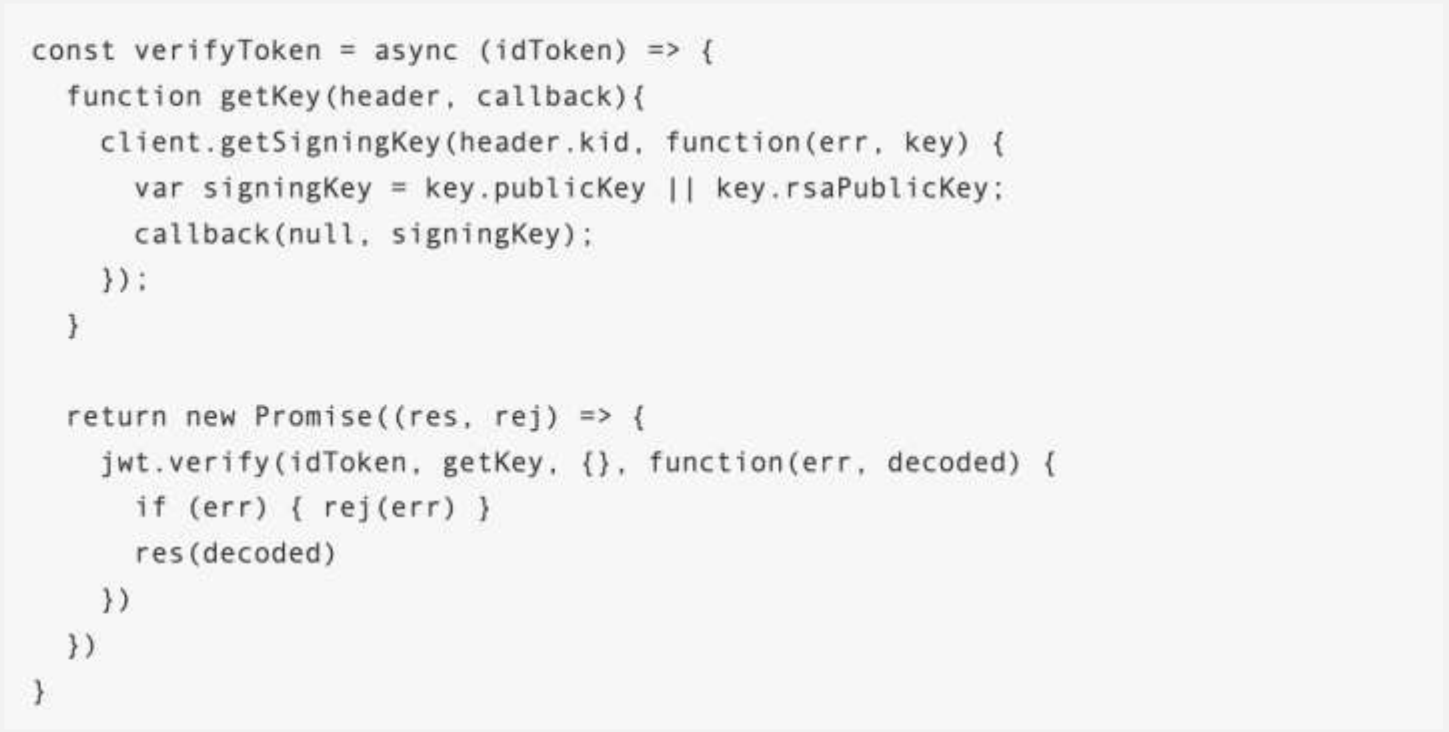
- The first function is createCognitoUser and is used when registering a new user in Amazon Cognito. The function looks as follows:

```
const createCognitoUser = async (username, password, email) => {
  const signUpParams = {
    ClientId: process.env.COGNITO_CLIENT_ID,
    Username: username,
    Password: password,
    UserAttributes: [
      {
        Name: 'email',
        Value: email
      }
    ]
  }
  await cognitoidentityserviceprovider.signUp(signUpParams).promise()
  const confirmParams = {
    UserPoolId: process.env.USER_POOL_ID,
    Username: username
  }
  await cognitoidentityserviceprovider.adminConfirmSignUp(confirmParams).promise()
  return {
    username,
    email
  }
}
```

The second core method is the login function that is used when registered users are authenticating. The code is as shown below

```
const login = async (username, password) => {
  const params = {
    ClientId: process.env.COGNITO_CLIENT_ID,
    UserPoolId: process.env.USER_POOL_ID,
    AuthFlow: 'ADMIN_NO_SRP_AUTH',
    AuthParameters: {
      USERNAME: username,
      PASSWORD: password
    }
  }
  const { AuthenticationResult: { IdToken: idToken } }= await cognitoidentityserviceprovider.adminInitiateAuth(params).promise()
  return idToken
}
```

- Finally, there is a verify token function. The contents of this function are as follows:



```
const verifyToken = async (idToken) => {
  function getKey(header, callback){
    client.getSigningKey(header.kid, function(err, key) {
      var signingKey = key.publicKey || key.rsaPublicKey;
      callback(null, signingKey);
    });
  }

  return new Promise((res, rej) => {
    jwt.verify(idToken, getKey, {}, function(err, decoded) {
      if (err) { rej(err) }
      res(decoded)
    })
  })
}
```

This function verifies an ID token that has been passed up with a request. The ID token given by Amazon Cognito is a JSON Web Token, and the verify token function confirms that the token was signed by your trusted source and identifies the user. This function is used in endpoints that require authentication to ensure that the requesting user has access.