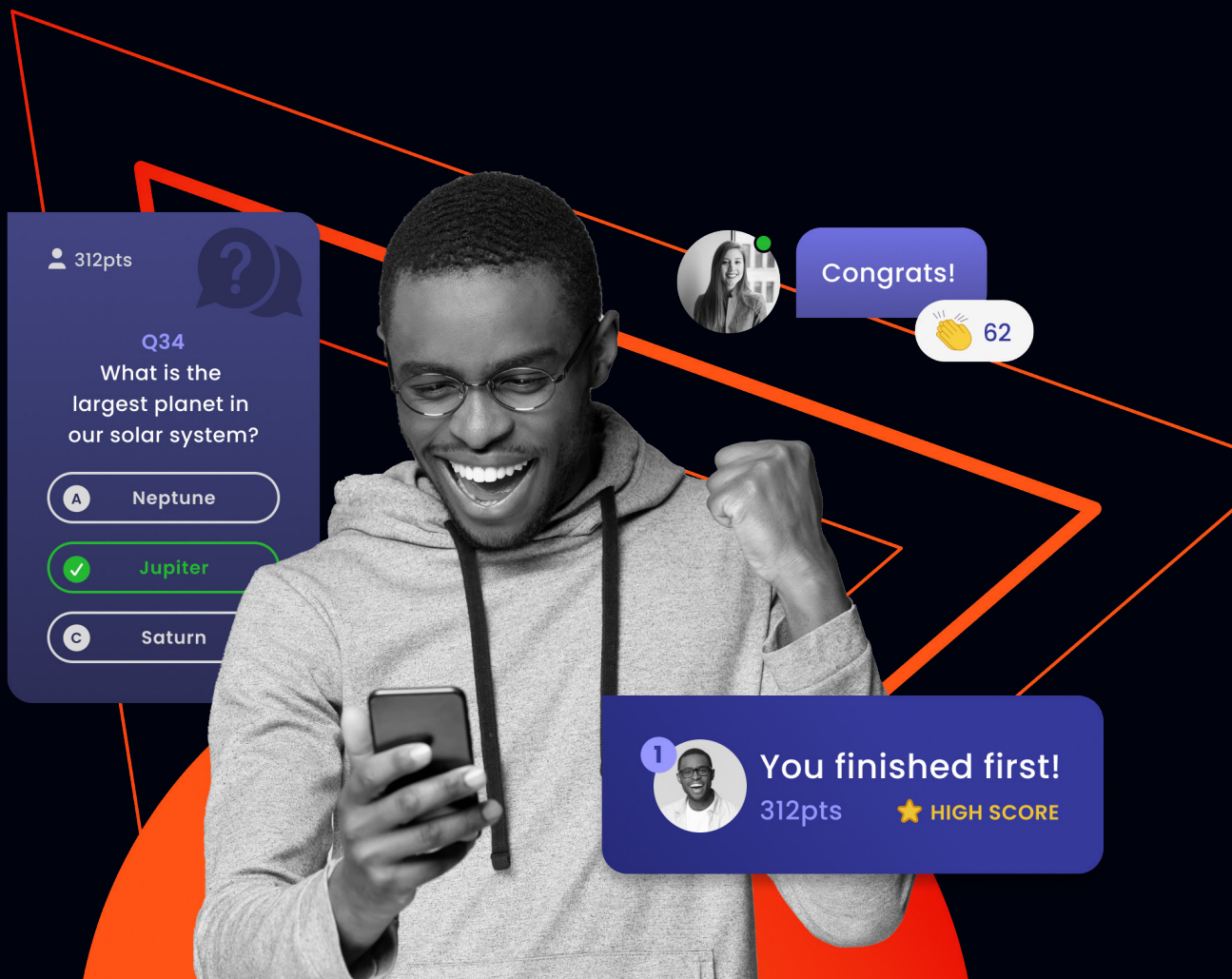
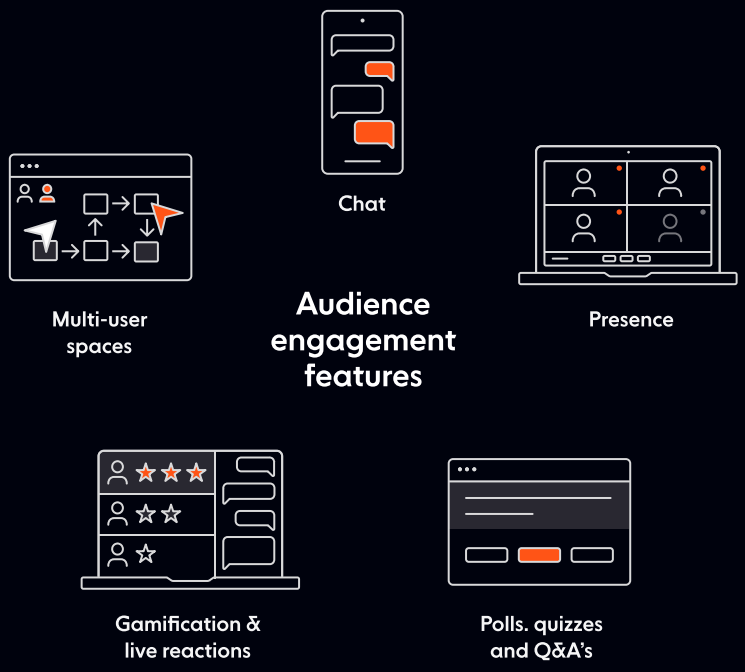
ably

WHITEPAPER

An event-driven architecture pattern for live in-app audience engagement features at scale



Live online audience engagement features are now an established part of any standard user experience to attract participants to applications and platforms. Examples include livestreams, with a host and a chat system for audience members to interact with each other instantly. Other typical features include chat or Q&A during a shared experience such as a Watch Party, as well as polls, quizzes, and leaderboards.



For audience satisfaction, engagement should be almost instantaneous, so any such solution needs low latency communication. To build new features alongside existing business logic, many engineering teams use an event-driven architecture, which slots alongside a traditional architecture, to meet the demands of realtime digital interactions.

Event-driven architecture has tremendous potential to serve use cases that need to process data immediately. It eliminates the need for blocking or constant polling and notifies application code whenever an event of interest occurs.

> This whitepaper will examine the high-level architecture required to support bidirectional low-latency messaging for in-app audience engagement. We'll use a multiplayer quiz app to explore the realtime message flow and implementation, but with minor differences, this could equally apply to other features such as chat, Q&A, or reactions.

Contents

An event-driven architecture pattern for live in-app audience engagement features at scale

Quizzes and gamification for audience engagement

Quizzes are a great way to engage an audience of, for example, students in a live lesson or team members in an online meeting. Participants receive quiz questions to answer via browser or app during the event, and the results and leaderboard are updated in realtime to encourage competition, shoutouts, and reactions.

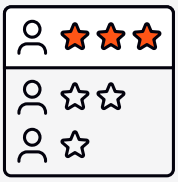
Be it a standalone quiz like Kahoot, or one that integrates into a slide deck like Mentimeter, the underlying technology behind these apps is similar and relies heavily on edge messaging architectures. Some popular products that offer multiplayer quizzes are listed include Mentimeter, Wooclap, Kahoot, Slido, and HQ Trivia.

Fundamental realtime features of a multiplayer quiz app

While each product puts a unique spin on its multiplayer quiz offering, we can assume some standard features as follows.

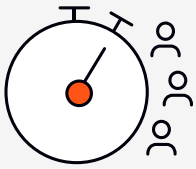
A host and a group of participants

The host is generally the curator of the quiz questions and runs the live quiz. The participants log into the specific quiz using an invitation link. Once in, they see the questions appear on their devices with options to register their answers.



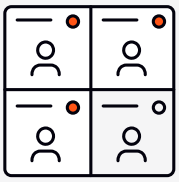
Live scores and a leaderboard

As participants answer the questions, they are scored based on accuracy and other factors, such as the time they take to respond. The host and the other participants can see a leaderboard updating with live scores.



Synchronized timer

Each question has a limited amount of time within which the participants need to choose an answer. This timer needs to run in sync for all the participants to ensure fair play.

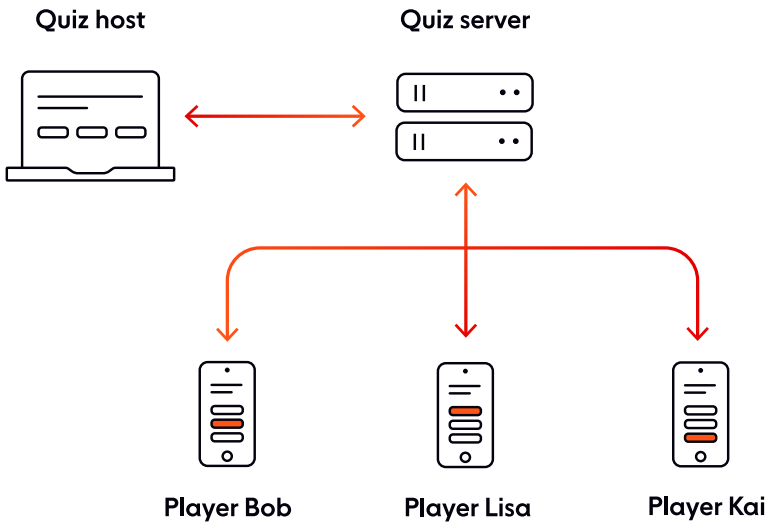


Presence

As these are mostly competitive quizzes, participants need to know how many people are present, who they are, and when their online status changes.

Architecture

A simplified view of the system that runs a multiplayer quiz for three participants consists of an app server and front-end clients for players and the quiz host.



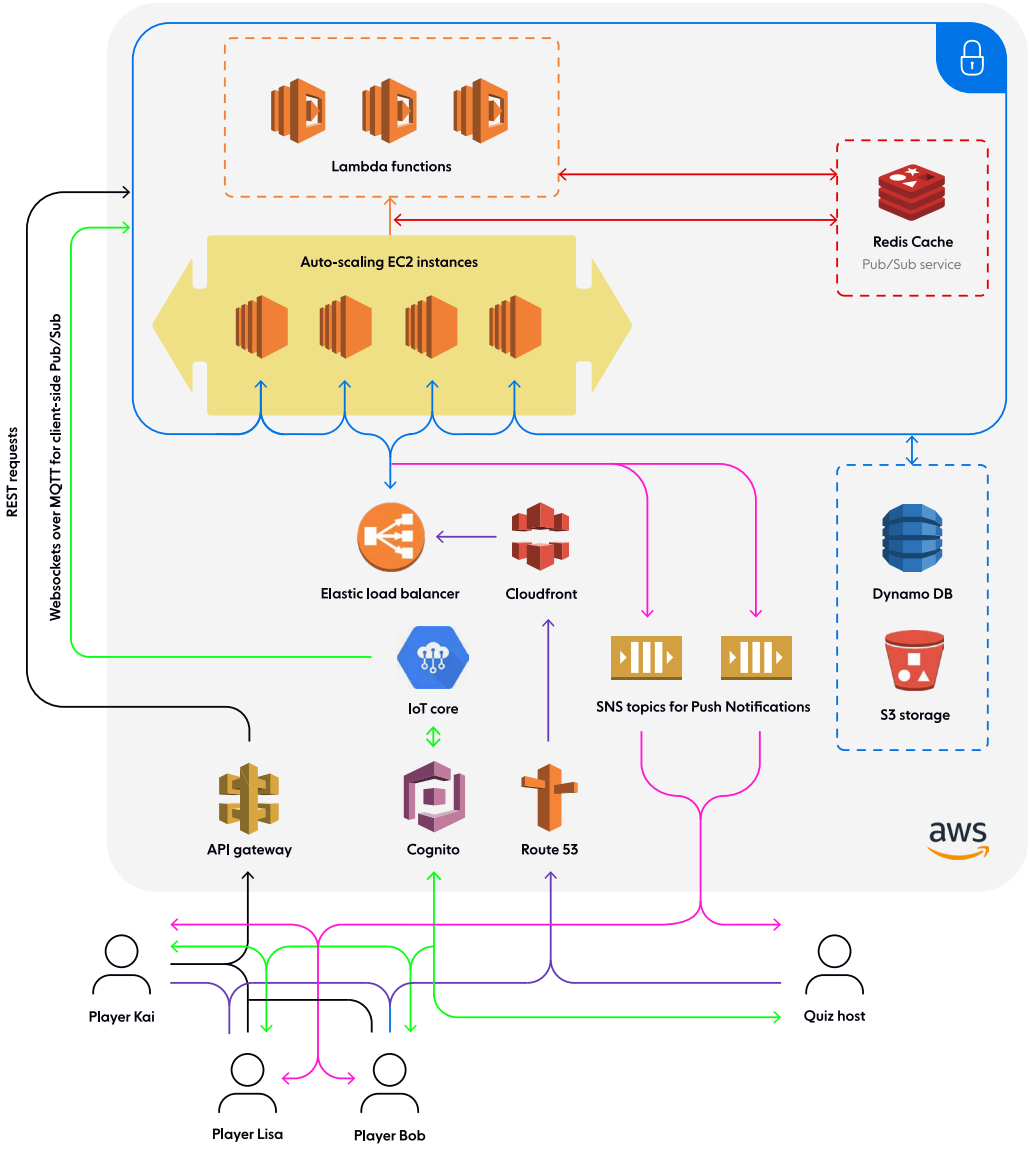
The app server publishes the questions, receives the answers, checks them, and computes the scores. Among its front-end clients, there is a host to control the flow of the quiz and participants that engage accordingly.

The most important aspect of this system is how various messages flow between these components. As shown by the two-sided arrows, we need a way for these components to communicate bi-directionally, in a stateful manner, possibly at high frequency.

> We'll begin with an example multiplayer quiz architecture implemented entirely on Amazon Web Services (AWS) before replacing some parts with Ably's edge messaging infrastructure to demonstrate how to remove some of the realtime complexity.

Multiplayer quiz architecture with AWS (single region)

Among the many web services offered by Amazon, we need the following components to run a live quiz globally, at scale.



The example shows a quiz host with three participants: Bob, Kai, and Lisa, who communicate with the backend server. We can assume the server-side logic is hosted on auto-scaling EC2 instances behind a load balancer.

Our front-end clients can access the quiz app via Route53 and Cloudfront before reaching the Elastic load balancer that decides which out of the available EC2 instances serve this request.

The backend VPC also communicates with the storage, i.e., S3 buckets and DynamoDB, to store and retrieve various pieces of information and assets.

We also have some Lambda Functions that will be triggered at different points in the server-side logic of the quiz. If our front-end clients also need to access the Lambda functions, they can do so via the API gateway.

This is fine for REST-based communication, which works based on stateless request/response cycles, but we also need publish and subscribe-based realtime communication capabilities. To enable those with Amazon Web Services, we'll need an Elasticache for Redis to serve as a global datastore with pub/sub messaging capabilities. Our backend servers hosted as EC2 instances will stay in sync with various updates within sub 250-millisecond latencies via this Redis cache.

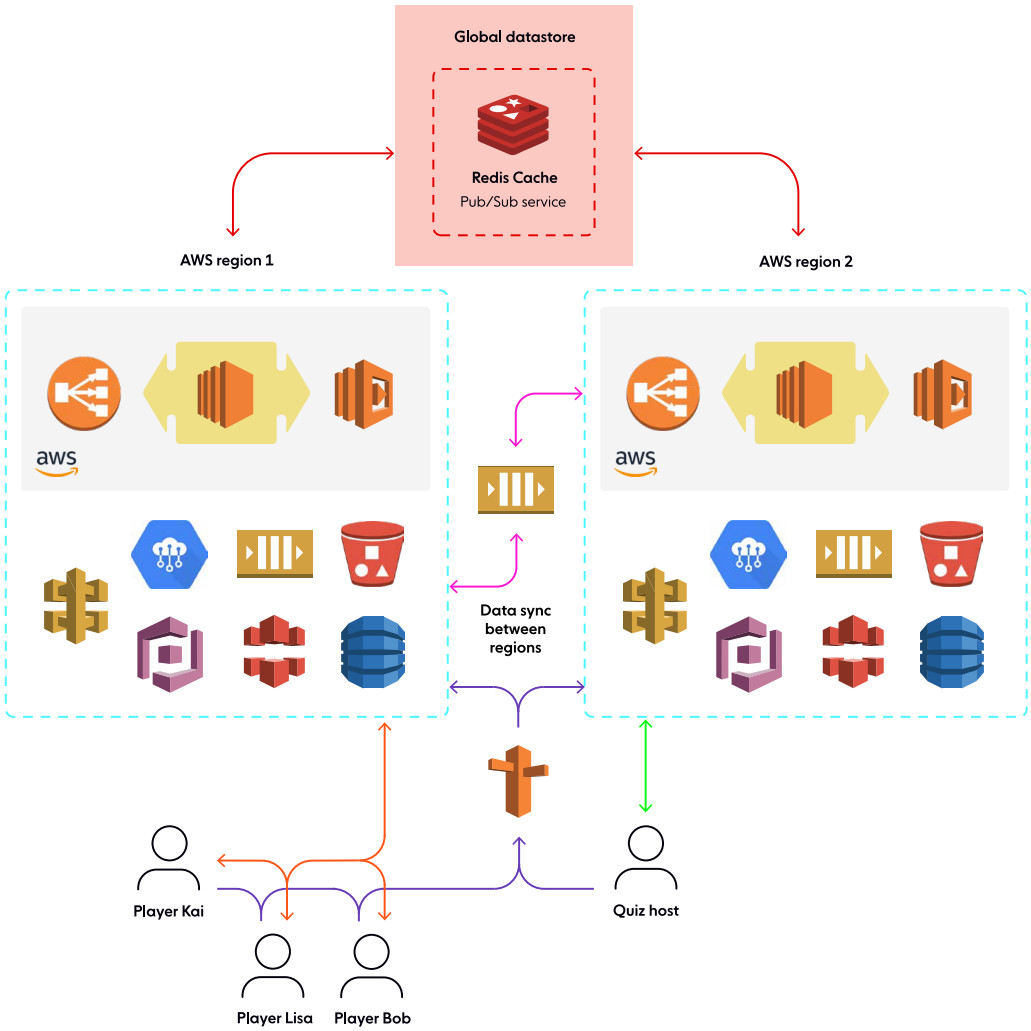
AWS recommends using the IoT core service to enable scalable pub/sub for our front-end clients, using WebSockets over the MQTT protocol. You also need an authentication mechanism for the front-end clients to reach the IoT core securely. The AWS Cognito service helps with this aspect.

While this enables app-based pub/sub messaging with the backend services, we still need something like SNS (Simple Notification Service) to push updates even when the app is not in use. This could be a mobile push notification to let our players know that a new quiz is starting, an email alert, etc. To understand the online status of the front-end clients, we need to implement that logic ourselves and publish messages when different events such as internet failures or app closures occur.

Multiplayer quiz architecture with AWS (multi-region)

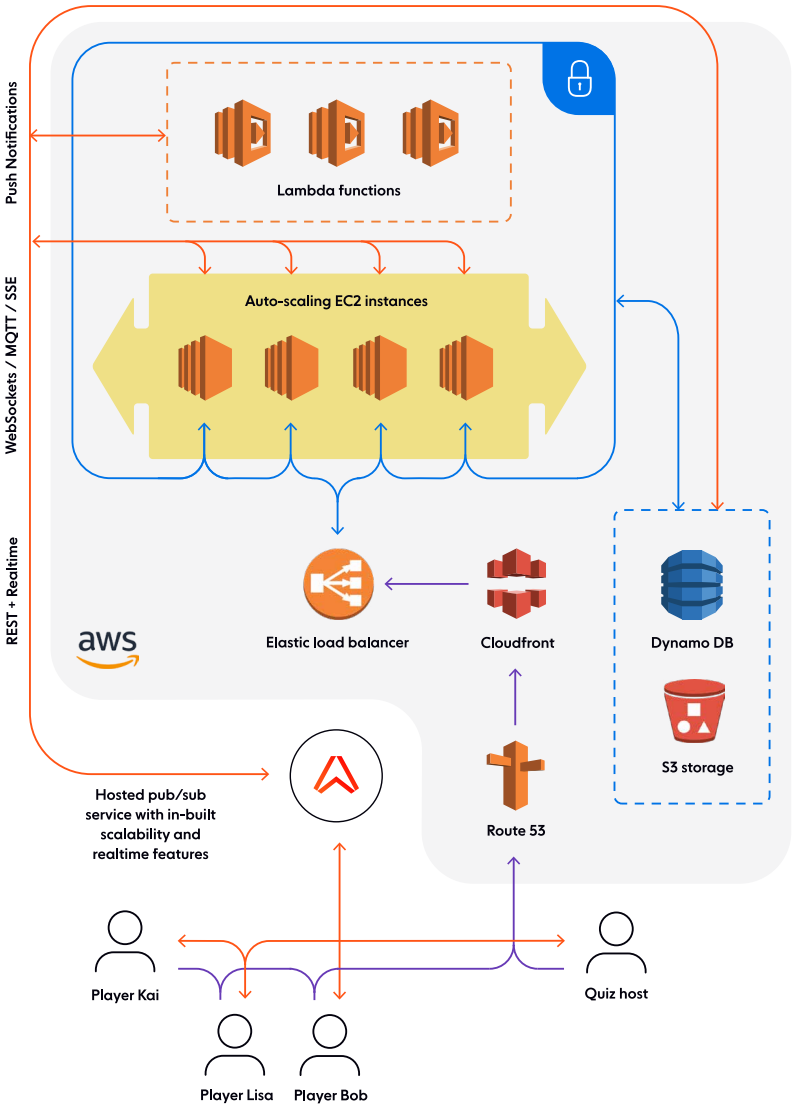
The architecture above considered a single AWS region, but for a live quiz to be global, you want all the participants to not only be synchronized, but also maintain sub 250-millisecond latencies. That goal is only achievable by ensuring that all clients are routed to a node nearest to them based on their geographic location.

To enable this, we'd need to replicate almost all of the services in multiple regions across the globe. This adds a non-trivial amount of architectural complexity, configuration, development, and maintenance to ensure these regions are synchronized at all times.



Multiplayer quiz architecture with Ably

Now, we can compare an architecture for a multiplayer quiz that uses Ably's edge messaging infrastructure to replace many of the AWS services listed above.



Out of the box, Ably automatically runs in multiple global regions; it does the smart routing internally without any configuration required by the developer. Ably can also communicate directly (via pub/sub and REST) with both back-end and front-end services to keep the backend in sync with itself and the front-end clients.

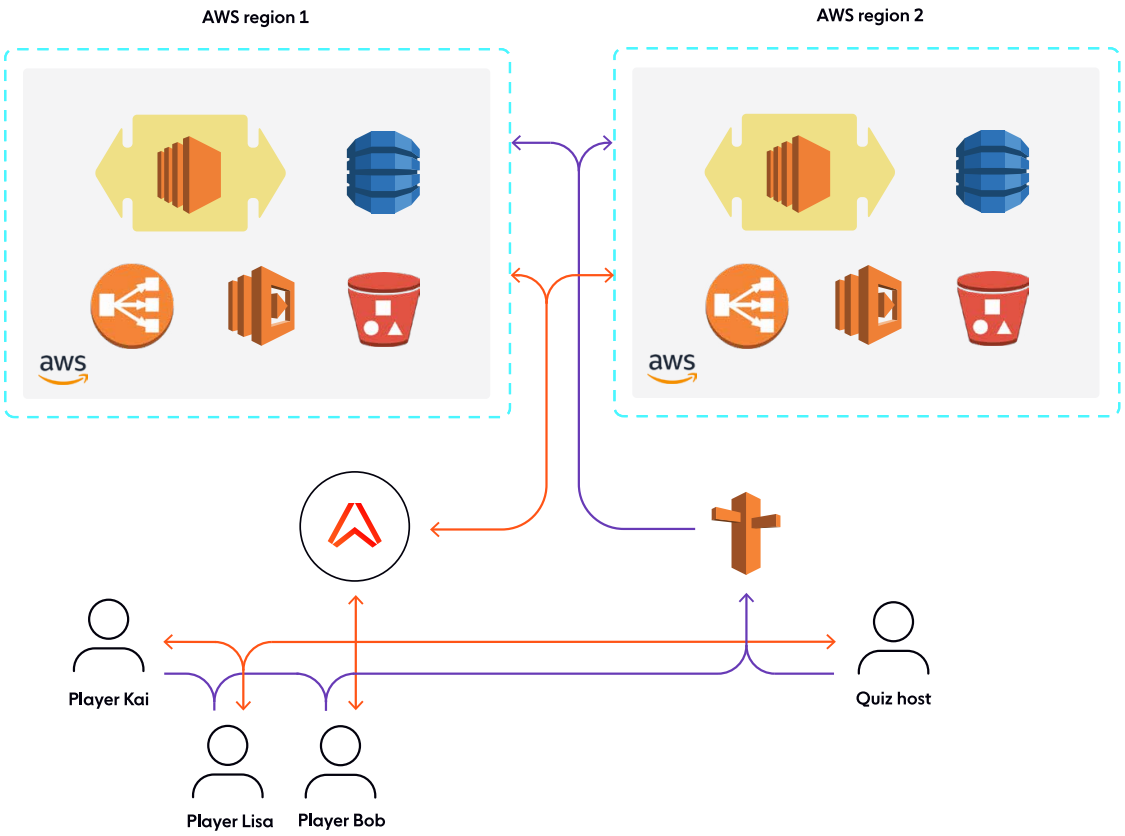
Consequently, we can remove the Elasticache for Redis, the IoT core, and the API gateway. We can also remove the SNS topics because Ably provides push notifications and direct integrations with third-party services via webhooks. This reduces the complexity of setting up these services, connecting them correctly, and maintaining them.

THE ABLY DIFFERENCE

Out of the box, Ably automatically runs in multiple global regions; it does the smart routing internally without any configuration required by the developer.

Ably makes it much easier to provide low-latency edge messaging at scale because it is a drop-in replacement for several AWS services. We can keep using AWS services to run the backend logic for the quiz, but we no longer need to deal with syncing various components in realtime as we'll be using Ably to do that. Ably handles all of the distributed systems challenges, including data-sync, availability, smart routing, and elasticity automatically, offering four pillars of dependability.

With Ably's hosted edge messaging infrastructure, you can enable realtime communication between any service via Ably and connect other third-party services or applications via webhooks, queues, or Firehose.



Ably channels

When you use Ably, you will send events into named channels. Clients attach to channels to subscribe to messages, and every message published to a unique channel is broadcast by Ably to all subscribers.

The Ably Realtime client library provides a straightforward API for publishing and subscribing to messages on a channel.

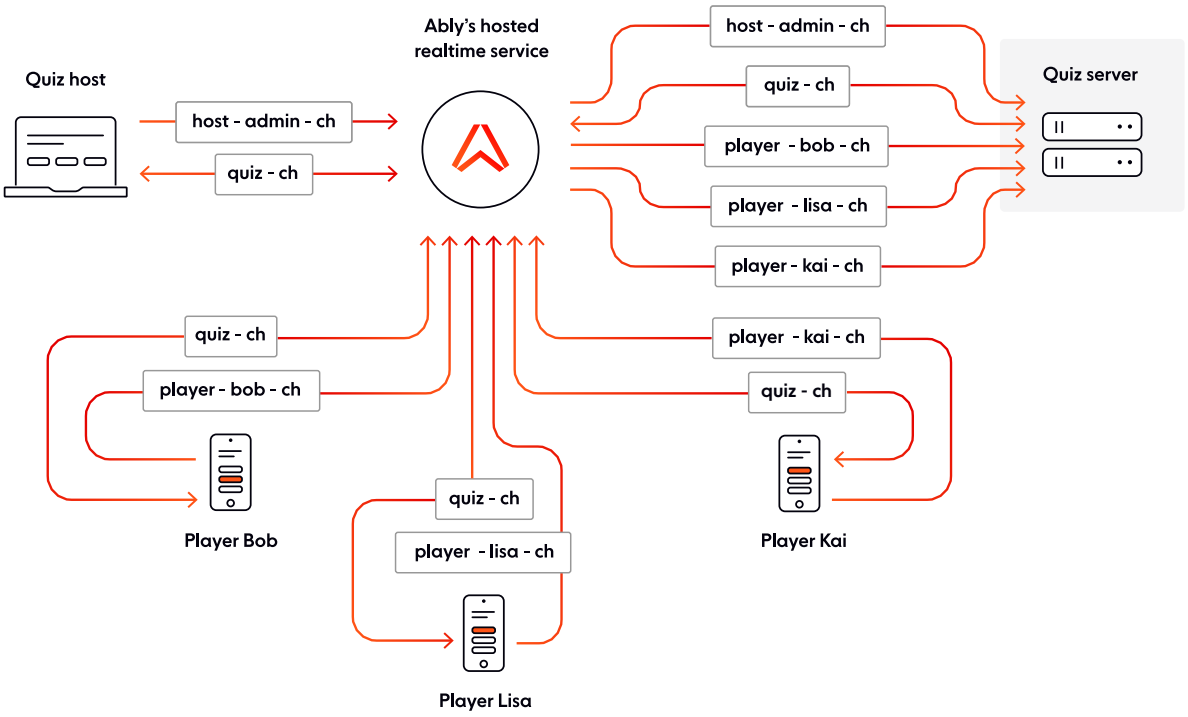
An example of using the Ably API to publish and subscribe messages over channels:

```
// API Keys should not be shared. The key here is intended
solely for this example.
const realtime = new Ably.Realtime('xVLyHw.OALQ2Q:aQMsViBg-
fUbTbtk5');
const channel = realtime.channels.get('classroom');
channel.subscribe(function(message) {
 alert(`Received: ${message.data}`);
});
channel.publish('example', 'message data');
```

Try it

Channel usage for a multiplayer quiz

For a multiplayer quiz app, here's how we could use channels:



Let's examine the categories of messages we'd like to stream between various components.

`host-admin-ch`

This channel is specifically meant for the host to trigger various events on the server like starting the quiz, showing the next question, etc. As you can see, no participants are part of this channel. We want only the host to have admin rights to trigger such events.

`quiz-ch`

This channel is the main quiz channel and transports the quiz-related data, including new questions, correct answers, or timer tick events.

`player-bob-ch`, `player-lisa-ch` and `player-kai-ch`

These three channels are dedicated to individual players to publish their answers to the questions in the quiz.

Of course, it's possible to have fewer or more channels, depending on the specifics of your use case. Another option is to have a single channel for all the players to publish their answers and use event names to let the server know which data belongs to which player.

Please note that this is a suggested architectural pattern. You can use it as a starting point and update the architecture to whatever suits you best, depending on the audience engagement use case you are implementing.

Presence: which users are online?

The presence feature allows us to see participants' 'online/offline' status. We can use the `quiz-ch` channel to manage the presence set as well. We need to have our host and participants enter the presence set as soon as they connect. They'll automatically disappear from the presence set if they quit the app or get disconnected for an extended period due to internet issues. There's no need to handle this logic yourself.

Here's a snippet to show how presence works:

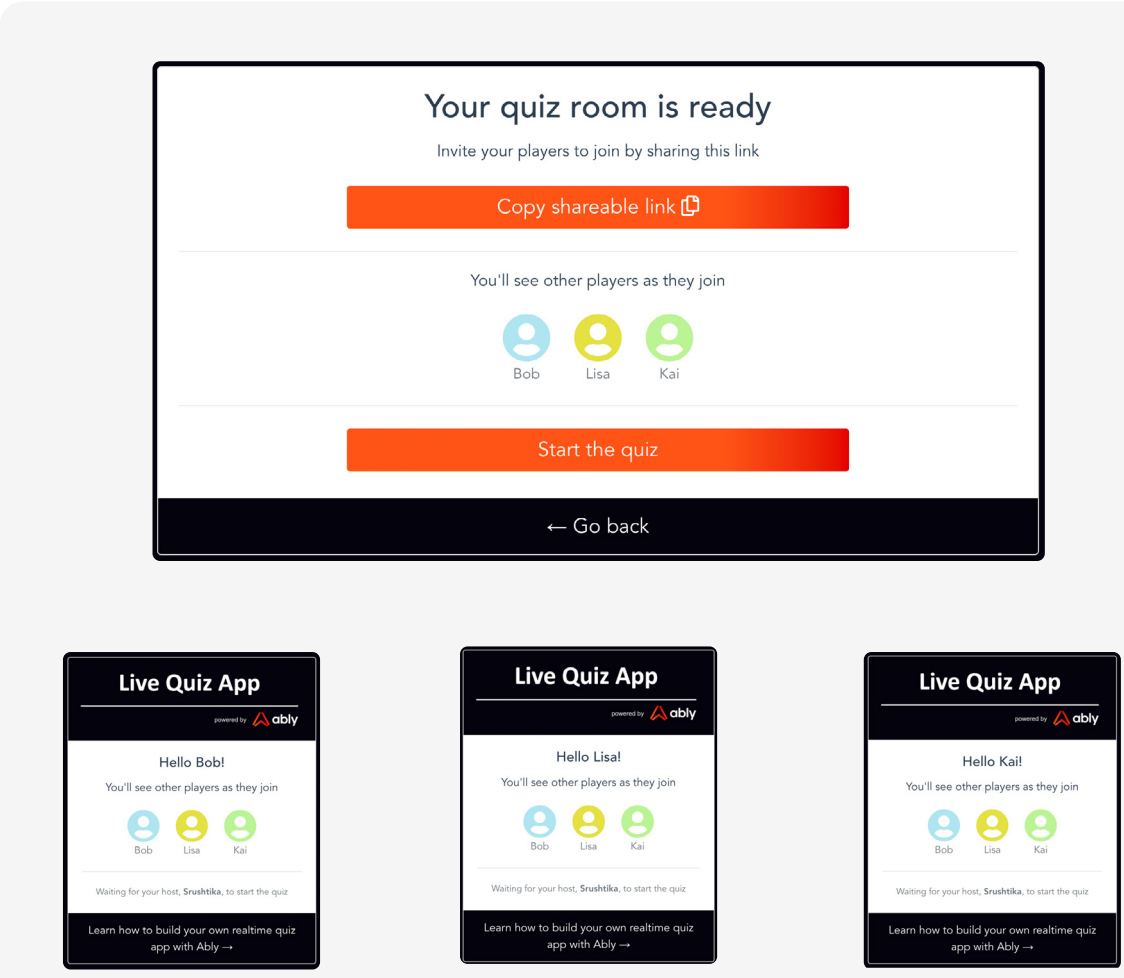
```
// API Keys should not be shared. The key here is intended
solely for this example.
const realtime = new Ably.Realtime({
 key: 'xVLyHw.OALQ2Q:aQMsViBgfUbTbtk5',
 clientId: 'bob' }
);
const channel = realtime.channels.get('classroom');
channel.presence.subscribe('enter', function(member) {
 alert('Member ' + member.clientId + ' entered');
});
channel.presence.enter();
```

Try it

In the demo quiz app (linked in the next section), you'll see that only the quiz server has subscribed to presence entries and leaves, so it can efficiently manage participants, live stats, scores, and leaderboard. It is possible to have all the components subscribe to presence events, but this can make it difficult to scale.

Starter kit and demo

To demonstrate everything discussed so far, we've built a starter kit in NodeJS and VueJS by following the multiplayer quiz architecture explained in this article. You can use it as a starting point and customize it to your needs or look at the open-source project to further understand it and build your own audience engagement solution from scratch.



This demo implements the worker threads feature in NodeJS to enable multiple quiz rooms, allowing various hosts to host their quizzes to groups of participants simultaneously. Check out the GitHub repo to learn more.

You can also try out the live demo yourself.

Extensions and integrations

You saw a basic architecture that allows you to easily build a multiplayer quiz app. However, you may need to add some more components to your system design in the real world.

Ably offers a dependable distributed serverless platform for edge messaging that can scale flexibly as needed. The implementation remains the same irrespective of the scale.

In terms of your product-specific custom architecture, you may want to add other components such as a database, maybe trigger a cloud function to perform some computation, or even stream messages to a third-party service. Ably provides easy ways to integrate with external APIs and services via webhooks, serverless functions, message queues, or event streaming. You can also use incoming webhooks to trigger a message on an Ably channel from an external service. (Think of a scenario where you allow participants to answer your quiz via SMS messages!).

Final thoughts

We hope this article gives you a headstart in building live audience engagement features at scale. We'd love to see your applications made with Ably, so feel free to tweet them to us @ablyrealtime.

References and further reading

Realtime challenges for audience engagement

The realtime web: evolution of the user experience

Building a realtime chat app with React, Laravel, and WebSockets

Scalable, dependable chat applications with Apache Kafka and Ably

Building realtime apps with Flutter and WebSockets: client-side considerations

About Ably

Ably is an edge messaging platform for developers. There's no infrastructure to provision or manage, just an evolving suite of SDKs and APIs that give you the freedom and flexibility to power shared live experiences with a few lines of code. Our mathematically modeled system design provides a global edge network that brings users closer to your app; unique data ordering and delivery guarantees ensure a seamless end-user experience; a legitimate 99.999% uptime SLA is underpinned by fault tolerant infrastructure; and instant elasticity enables effortless scale.

Brands like HubSpot, Toyota, and Webflow trust Ably to power shared live experiences like business-critical chat, order delivery tracking, or document collaboration for millions of simultaneously connected devices around the world.

Sign up for a free account

Get in touch

Visit
ably.com

Call
+44 20 3318 4689 (UK)
+1 877 434 5287 (USA)

Email
hello@ably.com