A Serverless Adventure

This supporting document will help to understand some of the details about the presentation at re:Invent 2018. It's required that you joined the session at re:Invent. All information here is constrained to the scope of this project, and tied to the status of the services available in September, 2018. Further service enhancements may bring more room for improvement and optimization.

Understanding the requirements

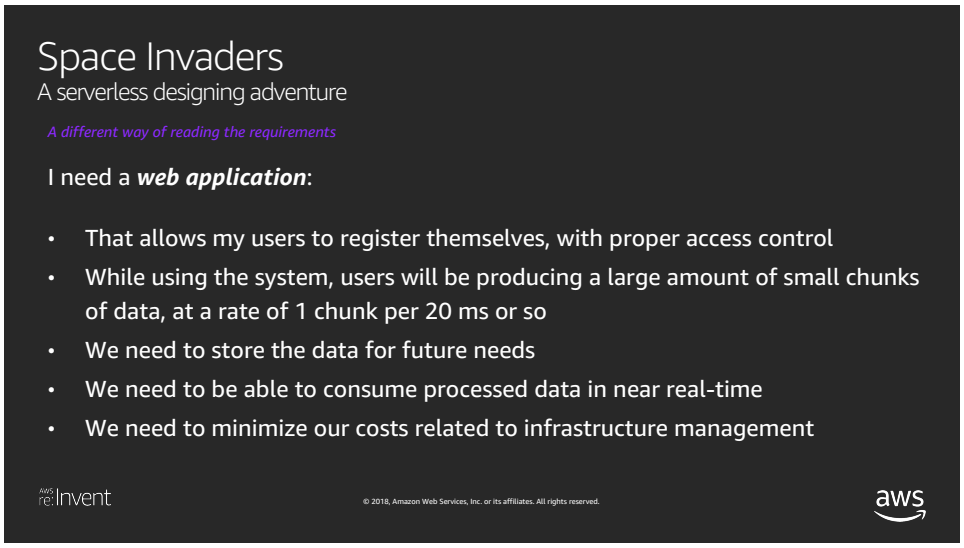
AWS Serverless SpaceInvaders is an application comprising two personas: The Gamer and the Manager.

The Gamer is the person playing SpaceInvaders against other gamers in that specific session. While playing, the gamer is sending updates about the score, and receiving a scoreboard view. On average, a gamer produces 53 events/second. These events correspond to score increases, missile firings, level changes, and life loss.

The Manager controls configuration, opens and closes a gaming session, and provides the full scoreboard view, such as the one below.

Nickname	Score	Shots	Level	Lives
USER131	200	1	2	0
USER92	198	38	2	0
USER47	198	48	3	0
USER82	196	32	3	0
USER69	195	9	3	0
USER147	195	41	3	0
USER59	194	46	3	0
USER106	192	40	3	0
USER24	191	23	3	0
USER114	189	21	3	0

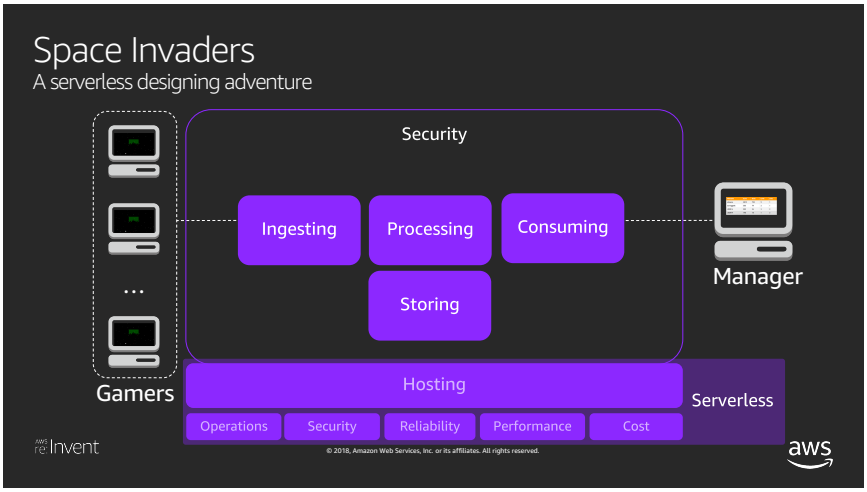
This is a system where some entities produce and upload data to the AWS cloud at a certain rate, where it is processed. The other entities view the results of this processing in real-time.



Think about how many different systems could be represented by these very same requirements, from retail stores, hospitals, to industrial IoT. All these systems require authentication and access control, data ingestion, data storing, data consumption—sometimes in real-time or near real-time.

What does “real-time” mean? Real-time is a physical-relativistic concept. It depends on the speed of the observed phenomena, on the characteristics of that phenomena that someone is interested in, and on the latency between the source of the phenomena and the observer. Let’s consider some examples: (1) NASA monitoring a

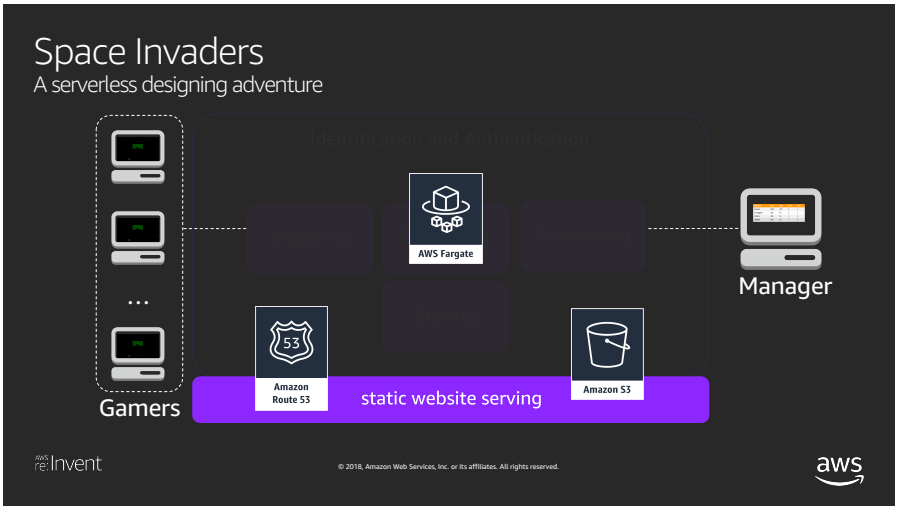
robot on Mars; (2) an engineer monitoring a formula 1 car; (3) a company monitoring a patient data; (4) physicists tracking particles resulting from a particle collision; (5) a system designed to acquire a car's plate image, interpret the numbers, check if the customer has enough cash for automatic payment to open the toll gate; (6) an IoT-based tracking system developed to keep your pet turtle from escaping (supposing that you have a turtle at home, of course). What are the implications of the expression "real-time" on each one of these systems? The implications will affect the decisions, the costs, and even the technology that will be used to implement the solution. For the example of the physicists, new technologies and appliances might need to be developed.



Our discussion is around selecting options in the domains of Hosting, Security, Ingestion, Consumption, Processing, and Storage, while complying with the guidance of the AWS Well-Architected Framework. For any system there are many different options for implementing the various components, especially on AWS where the breadth and depth of the technologies give many options to implement a system with specific needs.

Let's start by thinking about the possibilities for each one of these domains.

Website hosting



For this session, we used a static website, made by CSS, HTML, and Javascript files. The traditional combination of Amazon Route 53 and Amazon S3 is used. Amazon Route 53 is a fast, reliable, and highly available, globally distributed DNS web service. Amazon S3 is a highly scalable, multi-AZ storage service designed to deliver 99.999999999% durability. Both are highly cost effective and remove the heavy lifting of managing your own infrastructure.

Another option is AWS Fargate. Those companies already experienced with containers may want to deploy their websites using a fleet of containers. For this case, AWS Fargate is an option to run containers without managing servers or clusters. Pricing for Fargate is based on the effective usage of the combined computing power (vCPU) and memory power (GB) per second, billed at a 1-minute minimum.

Security

To support identification, authentication, and authorization needs, we're using Amazon Cognito and AWS Identity and Access Management (IAM).

Amazon Cognito

Amazon Cognito lets you add user sign-up, sign-in, and RBAC (role-based access control) to your web and mobile apps quickly and easily. Amazon Cognito scales to millions of users and supports sign-in with social identity providers, such as Facebook, Google, and Amazon, and enterprise identity providers via SAML 2.0

We are using Amazon Cognito to handle user sign-up, sign-in, maintaining user data, and to provide access tokens to the front-end application.

With Amazon Cognito User Pools, we assign users to groups to provide the authorization-side of the information system security. We have defined two groups—Gamer and Manager—and attached different IAM roles to each group. This way, when a user authenticates to the service, he or she is granted a role that provides an access profile, in accordance to the role attached to that specific group. For example, the Gamer has access to API Gateway, but has no direct access to other AWS services. The Manager has direct access to other AWS services, and to API Gateway.

Amazon Cognito User Pools is currently priced based on *monthly active users (MAUs*). A user is counted as a MAU if, within a calendar month, there is an identity operation related to that user, such as sign-up, sign-in, token refresh or password change. You are not charged for subsequent sessions or for inactive users within that calendar month.

Amazon Cognito User Pool features ***a free tier of 50,000 MAUs for users who sign in directly to Cognito User Pools*** and 50 MAUs for users federated through SAML 2.0 based identity providers. The free tier *does not* automatically expire at the end of your 12-month AWS Free Tier term, and it is available to both existing and new AWS customers indefinitely.

AWS Identity and Access Management

AWS Identity and Access Management (IAM) enables you to manage access to AWS services and resources securely. Using IAM, you can create and manage AWS users and groups, and use permissions to allow and deny their access to AWS resources.

IAM is a feature of your AWS account offered at no additional charge. You will be charged only for use of other AWS services by your users.

IAM Roles are used to configure the permissions for the groups defined on Cognito, and permissions for unauthenticated entities. This way, role-based access controls (RBACs) for the users in groups can be defined, controlling access to other AWS Services and your own APIs.

Considering the requirements for identification, authentication, and authorization, Space Invaders uses Amazon Cognito and AWS IAM at zero cost.

Think about the TCO of other alternatives.

Ingesting and Consuming Data

Decisions about real-time scale and sampling rate

Let's think for a moment about how the game works. When a gamer is playing, he/she is potentially providing data to the cloud at an average rate of 53 events/second. The payload is around 120 bytes, which is small and very similar to sensor-based workloads. We don't know yet how many players we want to have simultaneously in a session, but we would like to have as many as possible, potentially a global-scale game.

The experience that we are intending to provide is to have people playing in a room, or in remote places, with a big scoreboard projected in the room wall for everyone to be able to follow the competition, and for the players to be motivated to fight for the 1st position in the rank. We also want to provide a scoreboard for each player, so they can follow the game without needing to look to the big scoreboard being projected.

Considering these assumptions, *what is an acceptable concept of real-time for this scenario?* Do we really need to ingest every single event generated by a player? What happens if we miss 2 or 3 samples? Do we need to show all the players in the big screen? At what refresh rate? At each 20 ms? Do we need to show every player's rank? Or only the top 3? Maybe the top 10? This is a phase of requirements analysis that may affect our choices about the services, our decisions about the integration strategies, and will have direct implications on our cost.

By the way, these paragraphs could be the kick-off for an ideation or design thinking meeting, do you agree?

For this version of the game, we decided that:

- The game will publish the current player status at 300ms intervals (approx. 3 events per second). That means that we may lose intermediary changes in the player status but we don't consider those changes relevant for the scenario's objectives.
- The room scoreboard will comprise all the gamers, and be refreshed at each 1.5 seconds.
- The player-screen scoreboard will comprise only the top 10, and be refreshed at each 2 seconds.

The idea here is to implement the optimal sampling rate to provide a good user experience without flooding the back-end with unnecessary requests. It's a trade-off between cost and benefit, efficiency and frugality. What are your thoughts? Should it be less? Should it be more?

And we made another decision: we will constrain each session to 200 possible users in order to run the game under the limits of the free tier whenever is possible.

Mechanisms for Ingestion and Consumption

Most of the systems developed to support a mobile or a web application are constructed using an API as the front-end for the back-end resources. By using an API you can publish a well-defined contract that the application can rely on. We chose Amazon API Gateway as the entry-point for the communication to the cloud.

Amazon API Gateway

Amazon API Gateway is an AWS service that enables developers to create, publish, maintain, monitor, and secure APIs at any scale. You can create APIs that access AWS or other web services, as well as data stored in the AWS Cloud.

Why API Gateway? How much effort is needed to build a flexible API environment? Think about resiliency, scalability, security, configuration, integration, logging, versioning, environment separation, throttling, monitoring, billing, and more. Amazon API Gateway has all these features, and you only pay when your APIs are in use. There are no minimum fees or upfront commitments.

API Gateway provides very low latency and seamless integration with the AWS Services, allowing you to implement something that we could call "*codeless integration*".

There are also alternatives. You can build your own API service on top of a fleet of EC2 Instances, or over a fleet of containers managed by AWS Fargate. What would you need to do to ensure all the features API Gateway provides?

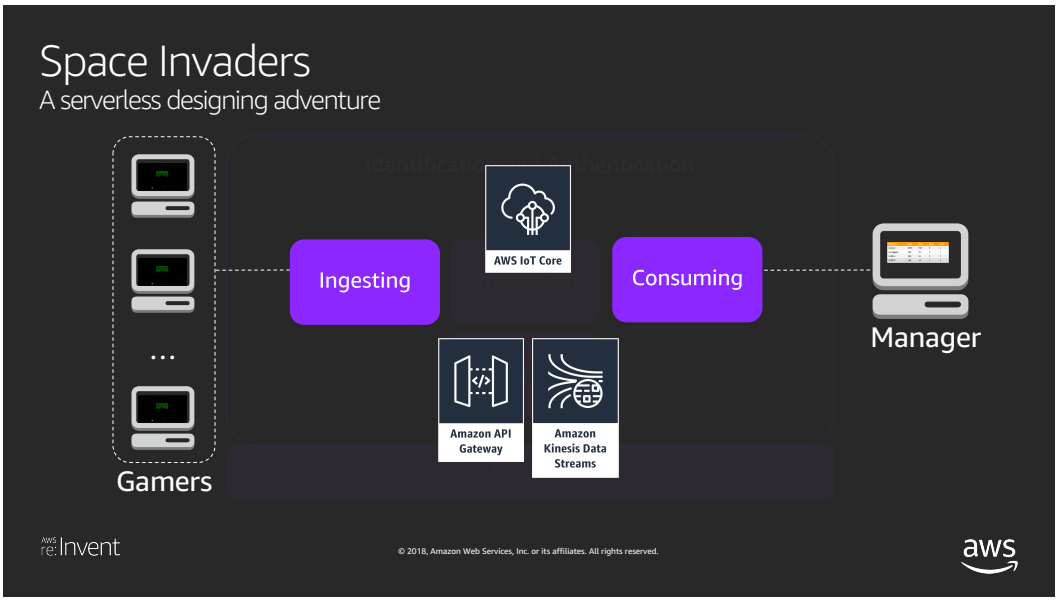
AWS Fargate

AWS Fargate is a compute engine for Amazon ECS that allows you to run containers without having to manage servers or clusters. With AWS Fargate, you no longer have to provision, configure, and scale clusters of virtual machines to run containers. This removes the need to choose server types, decide when to scale your

clusters, or optimize cluster packing. Fargate lets you focus on designing and building your applications instead of managing the infrastructure that runs them.

If you are in a company experienced in managing containers, or that has a long history of developing, deploying, and managing container-based API at scale, maybe Fargate will be the best option for you.

It's up to you to decide which option provides you the best composition in terms of TCO, speed, cultural fit, and support for innovation.



Why not AWS IoT?

Observing the ingesting and consuming dimensions of the game, the speed and size of data packets involved, why not use AWS IoT? AWS IoT Core lets connected devices easily and securely interact with cloud applications and other devices. AWS IoT Core can support billions of devices and trillions of messages, and can process and route those messages to AWS endpoints and to other devices reliably and securely.

You can use AWS IoT with Amazon Cognito, as Amazon Cognito can provide access tokens with permissions to access AWS IoT. However, AWS IoT is a better fit for managed devices. Managed devices are those which are built by you or by a partner of yours, providing structural limitations for one to have access to the credentials and mechanisms to push data to the cloud. If you grant access to AWS IoT to an unmanaged environment, like a web browser or other direct-ingestion service, you can expose the ingestion part of your system to someone who can submit payloads at rates or with content that are non-compliant to the requirements of your application, potentially requiring more controls in the back-end. This would increase your overall TCO and complexity for the solution. If Space Invaders was being played from a game console,

or from a mobile device through a mobile app, then AWS IoT could be reevaluated as a potential fit for the ingesting and consuming dimensions of the system.

Another option to consider is the feature of Basic Ingest, which removes the cost overhead of the message broker, allowing you to submit data directly to the AWS IoT Rules. With AWS IoT Basic Ingest for IoT-like systems you can reduce the ingestion costs at an order of 10x when compared to API Gateway. It's something to be considered if we manage the user's device. We must always remember that TCO is more important than cost, and security is our job zero. We need to think how IoT affects these dimensions for our applications, running on the browser or on mobile devices.

Processing Data

After API Gateway, before processing

Amazon API Gateway acts as an entry to the cloud. After receiving the payload, Amazon API Gateway needs to send it to the processing layer.

We could enable API Gateway to call the back-end directly. This means that for each API Gateway call, there will be a call to a processing layer. This is what we call a direct, coupled integration.

Decoupling is an important pattern because it gives us the opportunity of preserving the data even if there is a failure in the processing nodes. We could use queues, but for the nature of Space Invaders, where we have a *stream* of data coming to AWS, Amazon Kinesis Data Streams is a better choice.

Amazon
Kinesis Data
Streams

Amazon Kinesis Data Streams (KDS) is a massively scalable and durable real-time data streaming service. KDS can continuously capture gigabytes of data per second from hundreds of thousands of sources such as website clickstreams, database event streams, financial transactions, social media feeds, IT logs, and location-tracking events. The data collected is available in milliseconds to enable real-time analytics use cases such as dashboards, anomaly detection, dynamic pricing, and more.

Amazon Kinesis Data Streams has other advantages. It can natively store the data for a period of 24 hours (default) up to 168 hours (7 days). It can be natively integrated to AWS Lambda, such that the KDS service automatically and synchronously calls a Lambda Function at each second, providing it the ingested records. You can also configure how many records Lambda will receive, up to 10K records if the payload limit for Lambda is not exceeded.

So, with KDS we use two important features for Space Invaders: real-time data ingestion, and short-term storage for analytics.

Why not Amazon Kinesis Data Streams without API Gateway?

The answer for this question is similar to the reason we didn't select AWS IoT. We are running the player-side application on a web browser. All the access controls are tied to the permissions on the JWT token. This does not provide control for the rate of the requests, and the model for the payload. With API Gateway, we can control this.

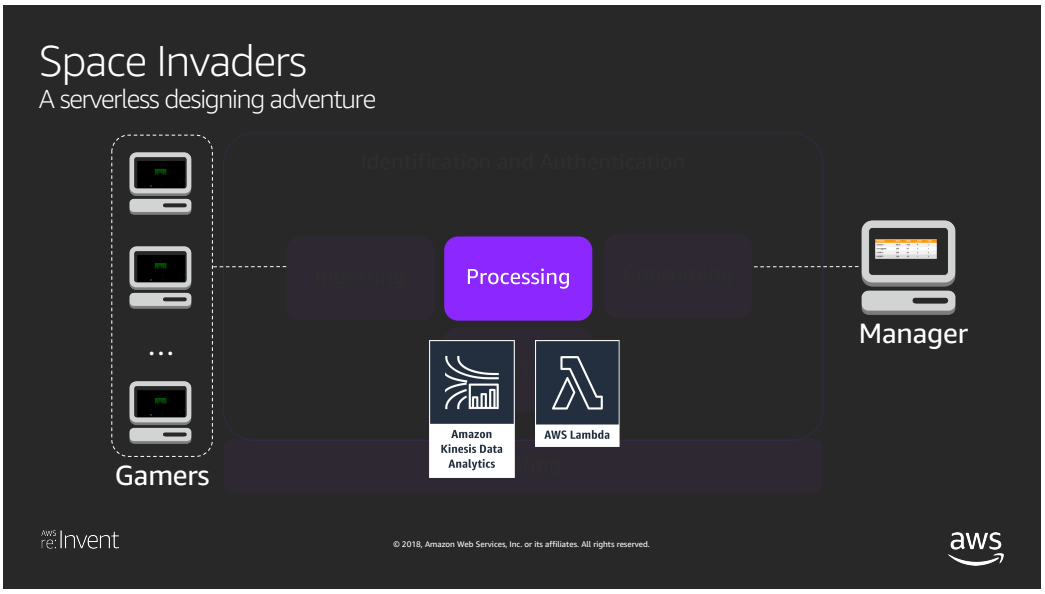
Again, this is relevant for unmanaged devices. If you have control of the end-user's device, then you can rethink this model.

At the processing layer

AWS Lambda

The most direct solution is AWS Lambda. AWS Lambda is a compute service that lets you run code without provisioning or managing servers. AWS Lambda executes your code only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the compute time you consume; there is no charge when your code is not running. With AWS Lambda, you can run code for virtually any type of application or backend service, all with zero administration. AWS Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code monitoring and logging. All you need to do is supply your code in one of the languages that AWS Lambda supports (currently Node.js, Java, C#, Go and Python).

AWS Lambda billing is time-based, for multiples of 100ms, based on the function runtime used and the configuration of memory and corresponding CPU power.



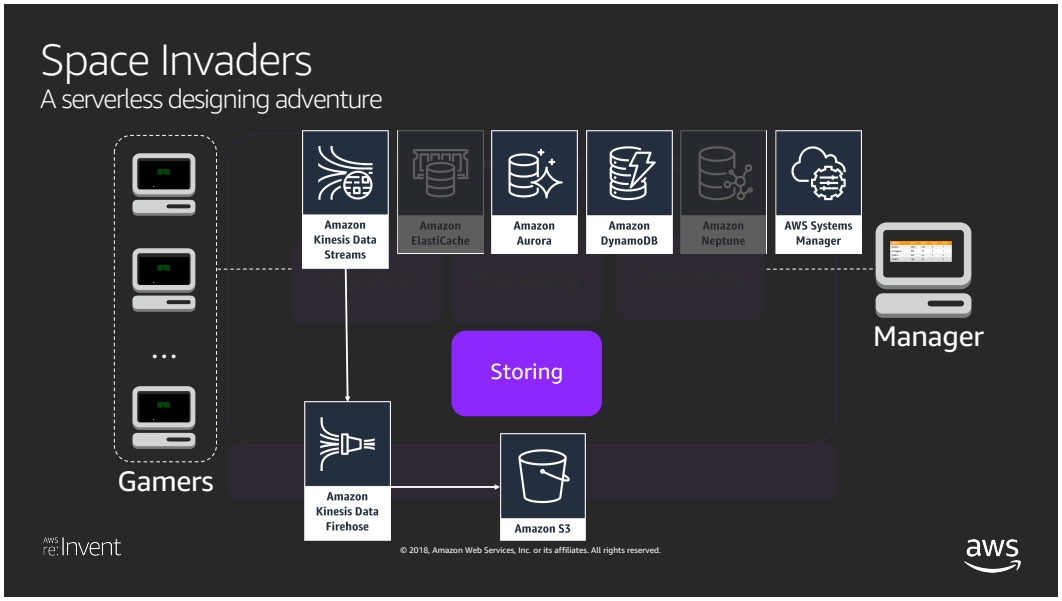
AWS Lambda is natively integrated to Amazon Kinesis Data Streams, allowing Lambda to consume records from KDS at a rate of 1 synchronous call per second. If you want a different behavior, you can implement your own mechanism for integration.

Another way to process data streams is through Amazon Kinesis Data Analytics. Amazon Kinesis Data Analytics is the easiest way to analyze streaming data, gain actionable insights, and respond to your business and customer needs in real time. Amazon Kinesis Data Analytics reduces the complexity of building, managing, and integrating streaming applications with other AWS services. SQL users can easily query streaming data or build entire streaming applications using templates and an interactive SQL editor. Java developers can quickly build sophisticated streaming applications using open source Java libraries and AWS integrations to transform and analyze data in real-time. You can use SQL or Apache Flink runtimes for Kinesis Analytics.

Both are interesting options for our serverless processing layer. So, why to stick to Lambda? First, we want to leverage existing knowledge and investments in Javascript development. Second, we want to leverage the free tier and cost advantages of AWS Lambda. The Lambda free tier includes 1M free requests per month and 400,000 GB-seconds of compute time per month. The Lambda free tier does not automatically expire at the end of your 12 month AWS Free Tier term, and is available to both existing and new AWS customers indefinitely.

Storage

There are many mechanisms for storing data on AWS. Some services are server-oriented, but fully managed by AWS, like Amazon Elasticache: a memory-based, sub-millisecond latency, managed in-memory database with Redis and Memcached flavors. Others, like DynamoDB and Amazon Aurora Serverless, are completely serverless, and you only need to worry about connecting and using it.



To select the appropriate mechanism for storing your data you need to think about the nature of your data (Is it temporal, document, time-series, or register oriented?) and about the access profile.

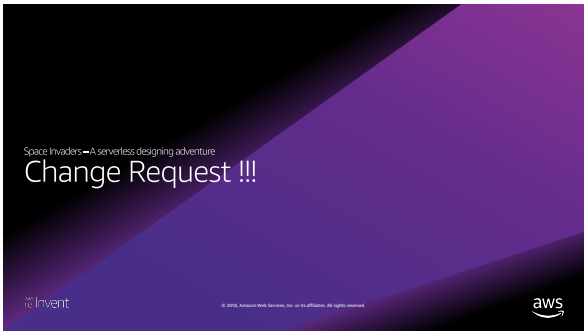
Space Invader session and scoreboard data is based on the SessionId and on the UserId (Nickname), and the events are sent timestamped, so we can track the changes through time. We have already decided to stream our data through Amazon Kinesis Data Streams (KDS), which is being used both as an ingestion mechanism and a short-term storage service. KDS stores all the events sent, and we can analyze it in real-time or after the ending of the session. But we need to select a database to store the scoreboard, both for in-game usage, and for long term storage.

Amazon DynamoDB

Considering the nature of the data, and access profile, we're choosing Amazon DynamoDB. DynamoDB is a cloud-scale, cloud-native, region-scoped, single-digit millisecond latency database with cost based on usage, and that provides a free-tier suitable for our Space Invaders sessions.

In addition to DynamoDB, we are also leveraging the features of Amazon Kinesis Data Firehose, which allows us to deliver data directly to different services on AWS, like Amazon ElasticSearch, Amazon Redshift, and Amazon S3. By moving all data from KDS to S3, we are storing our game data under eleven-nines of durability, in a region-scoped, highly-scalable service, and we can later analyze it using Amazon Elastic-Map Reduce or Machine Learning techniques to gain more insight about our game sessions and overall gaming experience.

Change request: Websockets



In our new scenario, the customer requires us to change the game experience. Currently, all the gamers join the game in their own time, not necessarily playing at the same time. But the customer wants to have the option of triggering all the players to start at the same time.

This new requirement has some relevant implications. First, we need a mechanism to communicate directly to the end-user devices, to the Space Invaders' Gamer Application. This requires some kind of push notification coming from the Space Invaders' Manager Application to all gamers, and joining them to the game at the same time.

When we have all gamers joining together, we will have a huge spike in the API, and consequently in the remaining layers. We need to be ready to respond to these spikes with proper initial sizing of Amazon Kinesis Data Streams and Amazon DynamoDB.

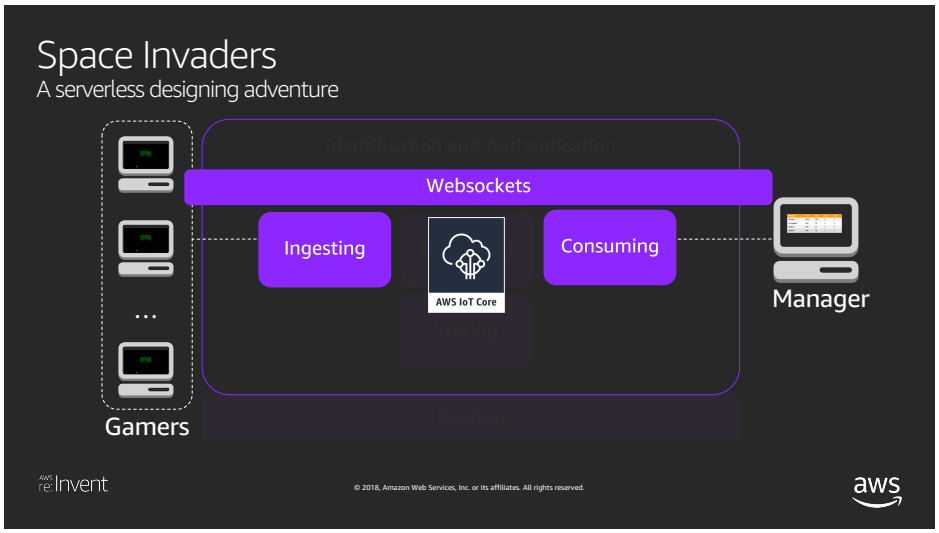
To implement the requirements for push notifications we can think about some services already available that can implement push notifications.

Amazon SNS allows you to send push notifications using system-to-system messaging, having a Lambda Function as subscriber, an Amazon SQS queue as subscriber, or a HTTP/S endpoint as subscriber. It also provides mechanisms for user notifications using a mobile application as subscriber.

Amazon Pinpoint is another service that provides push notifications to mobile applications using Firebase Cloud Messaging (FCM), Apple Push Notification service (APNs), Baidu Cloud Push, and Amazon Device Messaging (ADM).

But Space Invaders is running in the browser and is not a mobile-application. It does not provide an HTTP/S endpoint in the current browser implementation.

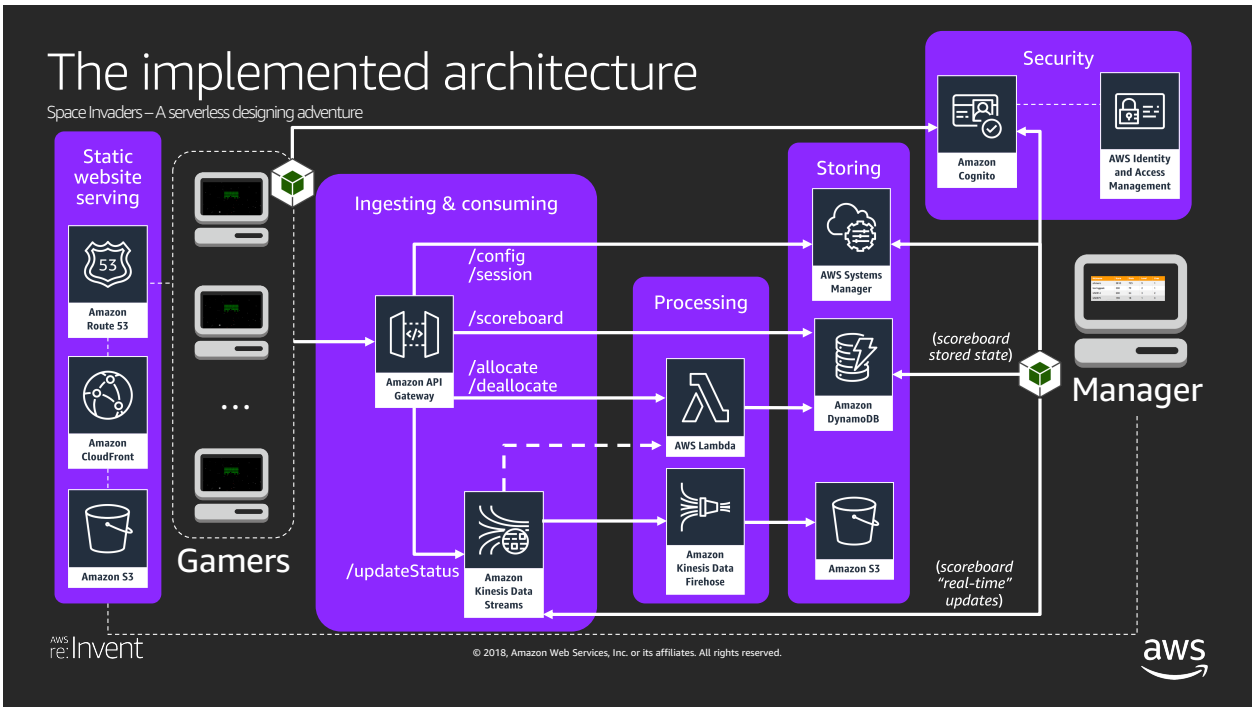
AWS IoT Core can provide a websocket protocol interface, so it is a viable option for implementing this requirement in a serverless way.



This means we may decide to use AWS IoT for all the Ingesting and Consuming domains of the system, or just use it for this particular feature. The decision will again rely on the level of trust over the end-user's devices.

This feature was not implemented in the game currently. As soon as the code is available, it will be a good opportunity to do it yourself, or by joining one of our Space Invaders workshops that will be delivered by our Partners.

The Space Invaders architecture for now



This is the implemented architecture for the first version of the system, aiming to comply with all customer's requirements.

We are using S3 to host a static website on top of AWS. Amazon Route 53 along with Amazon Certificate Manager are used to provide to provide scalable, highly available, and easy-to-use serverless mechanisms to procure, configure, and install HTTPs-backed domains. For our case, spaceinvaders.ninja was the choice.

AWS SDK for Javascript is being used for both the Gamer and the Manager applications, and provides us mechanisms to access AWS APIs. Both are using it to access Amazon Cognito, and the Manager, as a managed application, is using it to have direct access to the service APIs.

Amazon API Gateway as we can see is the front-end for the Gamer application, acting as the front door for other AWS Services.

Some resources of the API are directly integrated to other AWS Services without any code, just configuration.

The **/config** resource is directly integrated to AWS Systems Manager, and retrieves from the Parameter Store the configuration for the Game, getting the necessary information for the Gamer application to connect to Amazon Cognito, to provide the sign-up and sign-in features.

The **/session** resource is directly integrated to AWS Systems Manager, and provides a mechanism for the Gamer Application to identify if there is a session opened and get the session details.

The **/scoreboard** resource is directly integrated to AWS DynamoDB and doesn't require any coding for integration. This resource provides the "top players" scoreboard that is shown on the gamer's screen.

The **/allocate** and **/deallocate** resources are integrated to AWS Lambda, where code function checks if there are available seats to allocate to the gamer. It also deallocates the gamer when the browser window is closed. Some improvement will be necessary in the future: what's the best way to identify and automatically deallocate idle users?

The players post their status via the **/updatestatus** resource, moving the data to the Amazon Kinesis Data Streams. As we already stated, Amazon Kinesis Data Streams acts both as a short-term storage, and as a mechanism to deliver the ingested data both to S3, for long-term storage, and to Amazon Lambda, for real-time processing of the scoreboard. A Lambda Function is triggered once per second to read the scoreboard from DynamoDB and update it with the latest scores from the users. The Lambda function is configured to ingest 1000 records per second, which is more than enough to handle our intended audience of 150-200 people per session.

Finally, as the Manager is a trustworthy entity, the Manager does not rely on API Gateway for enhanced control. The Manager application is using the AWS SDK for Javascript to access the services directly.

The architecture for tomorrow's needs

We want you to remember that the system architecture for today is not necessarily the system architecture for tomorrow. As requirements evolve, the volume of transactions and data grows, and new services and features are available, we are pushed to move forward with architectural improvements and adaptations, and changes may appear in each one of these layers. With AWS, you can quickly experiment with new arrangements and evolve your architecture without incurring upfront and long-term costs.

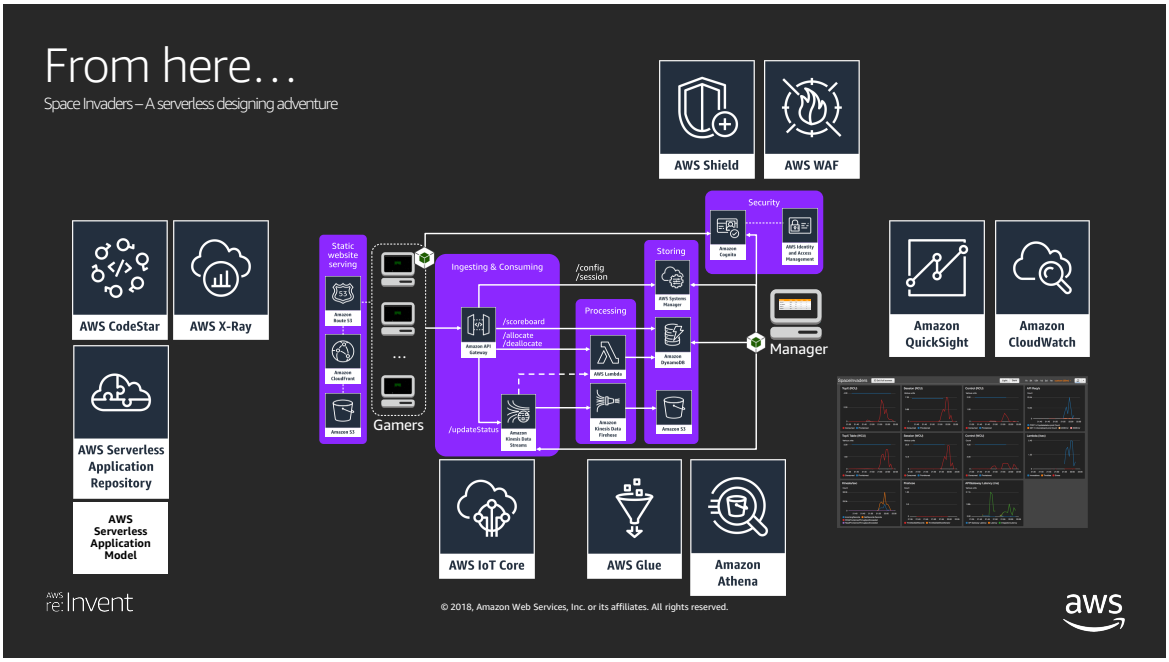
Suppose that we have a growth on the size of the payload, or an increase in the sampling rate, or that we need to move to a global-scale game, with millions of players in a session. What are the implications to our architecture? What should we change?

Suppose that we increase the number of users, immediately increasing the ingestion record-rate and volume for Kinesis, and one shard will not be able to handle the load. We need to design a mechanism for scaling up and down Kinesis shards, and for load-balancing the gamers to these shards as they are enrolled to the game, and when they sign-out from the game.

Suppose we increase the size of the payload. If it goes beyond 128Kb then we will need to drop the possibility of using AWS IoT, because this is the limit for the payload.

So, possible changes in the sampling rate, payload size, number of simultaneous users, or other functional and non-functional requirements will require us to evaluate the impact over the limits and scaling mechanisms of the selected services.

Where to go from here?



The design of the core of the game is just a starting point. We can use Amazon CloudWatch and Amazon Quicksight both for monitoring the infrastructure or the game sessions. We can use AWS Shield and WAF to further improve our security measures, use AWS Glue and Amazon Athena as mechanisms to analyze the ingested data stored on S3, and AWS CodeStar family, including CodePipeline, CodeCommit, CodeBuild, CodeDeploy, for a serverless approach for automating our development pipeline. We can even use AWS AI services to predict the winner, or to create an AI-based gamer. The possibilities are many, and with AWS there are no reasons to refrain from experimenting and innovating.

Explore, and go build!

Thank you!

Thank you for investing your time at re:Invent with us. If you are a customer or a partner interested in using Space Invaders as a starting-point to play, understand, and exercise serverless architectures, please contact us to get more input, and to provide feedback.

All the participants of the session will receive a communication when the code is made available. This will happen by the end of January 2019.

Fabian Da Silva

Partner Solutions Architect / US Northeast Region

fabisilv@amazon.com