

Design for Consent Management Platform

Introduction

The Consent Management Platform provides scalable, open-source services designed to empower companies to efficiently manage and enforce customer data privacy preferences.

In today's regulatory landscape, compliance with strict privacy regulations such as the EU's General Data Protection Regulation (GDPR) has become an imperative. By embedding privacy protections across business processes, organizations can stay ahead of evolving regulations and build customer trust.

The Consent Management Platform offers a suite of tools tailored to meet evolving business needs:

- **Customizable Consent Management:** Manage and customize consent agreements between users and services, ensuring compliance with regulatory requirements and company policies.
- **Querying Capabilities:** Query service- and user-specific consents to enforce individual preferences, enhancing data privacy governance.
- **Comprehensive Audit Trail:** Maintain a detailed audit history of all consent-related actions, including creation, updates, expiry, and revocations. This provides transparency and accountability for both businesses and their customers.

Benefits:

- **Scalability:** Our platform offers scalable solutions suitable for businesses of all sizes, accommodating growth and evolving needs.
- **Security:** All data is encrypted in transit and at rest, and stored using zero-access encryption so that we cannot read any of your data or share it with third parties.
- **Transparency:** All our service and infrastructure code is open-source and publicly hosted on GitHub for customers to view and suggest improvements to.

Requirements

Business requirements

P0 requirements

1. As a company, I need to know what consent my users have granted to each of my services, in order to customize their experience to their privacy preferences.
2. As a company, I need to be able to programmatically read/write/list consent records for a given {service, user} pair, so that I can integrate my own websites and backend services to enable my users to manage their consent.
3. As a company, I need to be able to answer whether a given user has granted a service a specific type of consent, in order to enforce their privacy preferences in my services.
4. As a company, I need to maintain a historical record of all consent changes made over time and be able to search this history for a given { service, user } pair.
5. As an end user, I need to be able to view all my active consents for a given service.
6. As an end user, I need to be able to revoke or update any granted consent to a given service.
7. As an end user, I need to be able to set time limits on consent granted to a service.
8. As an end user, I need to be able to revoke all consents granted to a given service.

P1 requirements

1. As a company, I need to be able to search for all users with a specific type of active consent to a service.
2. As a company, I need to be able to search for all consent history associated with a given service.
3. As an end user, I need to be able to view all previously granted consents to a given service which may have expired or been revoked.

P2 requirements

1. As an end user, I need to be able to view my history of granted/expired/revoked consents.
2. As a company, I need to be able to track metrics for aggregate user consents to my service.

Technical requirements

1. The platform must provide API access to onboarding companies to access/manage consent data and view consent history.
2. The platform must provide p95 API latency SLAs within:
 - a. 50ms for retrieving a page of consents for a {service, user, consent type} tuple
 - b. 50ms for retrieving a page of consents for a {service, user} tuple
 - c. 50ms for creating/updating a consent record

- d. 50ms for retrieving a page of historical consent updates for a {service, user} tuple
 - e. 50ms for retrieving a page of historical consent updates for a user
- 3. The platform must automatically update active consents with expiryTime values to have an expired status within 24 hours of their expiryTime.
- 4. The platform must support peak traffic of:
 - a. 2.5k transactions/second retrieving consents for {service, user} or {service, user, consent type} tuples
 - b. 100 transactions/second creating/updating consents for {service, user, consent type} tuples
 - c. 100 transactions/second retrieving consent history for {service, user} tuples
- 5. The platform must provide p95 data freshness SLAs of:
 - a. 300ms for Consent Management API responses
 - b. 5s for Consent History API responses
- 6. The platform must support 250 GB of consent data, and 1 TB of consent history data

Security requirements

- 1. [Availability] The platform must remain continuously available including during software maintenance activities, maintaining an overall annual availability SLA of at least 99.9%.
- 2. [Availability / Data Recovery] The service must maintain automatic point-in-time backups of all data that can be restored upon request.
- 3. [Integrity/Auditability] The service must never delete consent data, only change its status.
- 4. [Integrity/Auditability] The service must never delete events from consent history, only add.
- 5. [Confidentiality] No company should be able to access another company's service consent data.
 - a. Will enforce this by creating a new cloud services account for each onboarding company, with their own deployed instance of the platform.
 - b. Will enforce access controls so that only host platform administrators can access and maintain backend resources, and the client company's services and/or accounts are authorized to access API endpoints on an as-needed basis through IAM roles.
- 6. [Confidentiality] No user should be able to view consent data associated with other users.
- 7. [Confidentiality] No unauthorized user or service should be able to access the consent APIs.

Technical design

System architecture

Note: Source diagrams maintained at <https://github.com/Consent-Management-Platform/consent-management-design>, built using DrawIO at <https://app.diagrams.net/>.

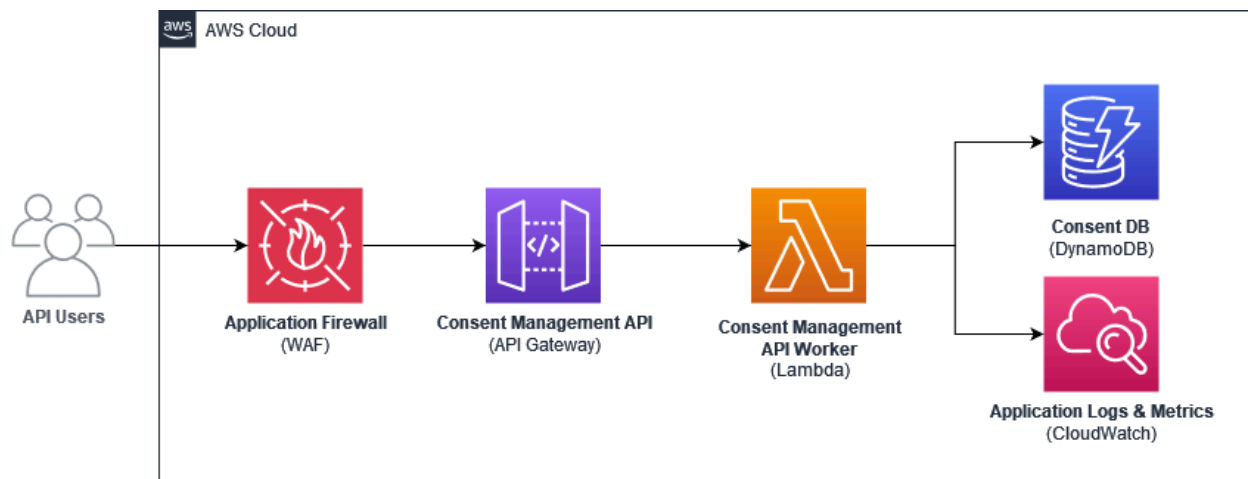
Design alternatives considered and trade-offs are included in appendices.

Consent management API

Consent management API architecture diagram

The Consent Management API will route API traffic through a Web Application Firewall (WAF) that will filter out common forms of malicious traffic to protect the backend service and maintain availability for legitimate users throughout common denial of service attacks.

The WAF will pass legitimate traffic through to the API Gateway, which will provide API authentication/authorization, per-client rate-limiting, and basic request input validation before passing through to the appropriate Lambda function, which will query backend data stored in DynamoDB and emit application logs and metrics to CloudWatch.



Technologies

- Database: AWS DynamoDB
 - Good fit for use cases, significantly lower cost than other options, and higher availability and lower maintenance than SQL-based databases
- Compute service: AWS Lambda while peak traffic is < 1k requests/second
 - Simplest and least expensive option while observed traffic is < 1k requests/second

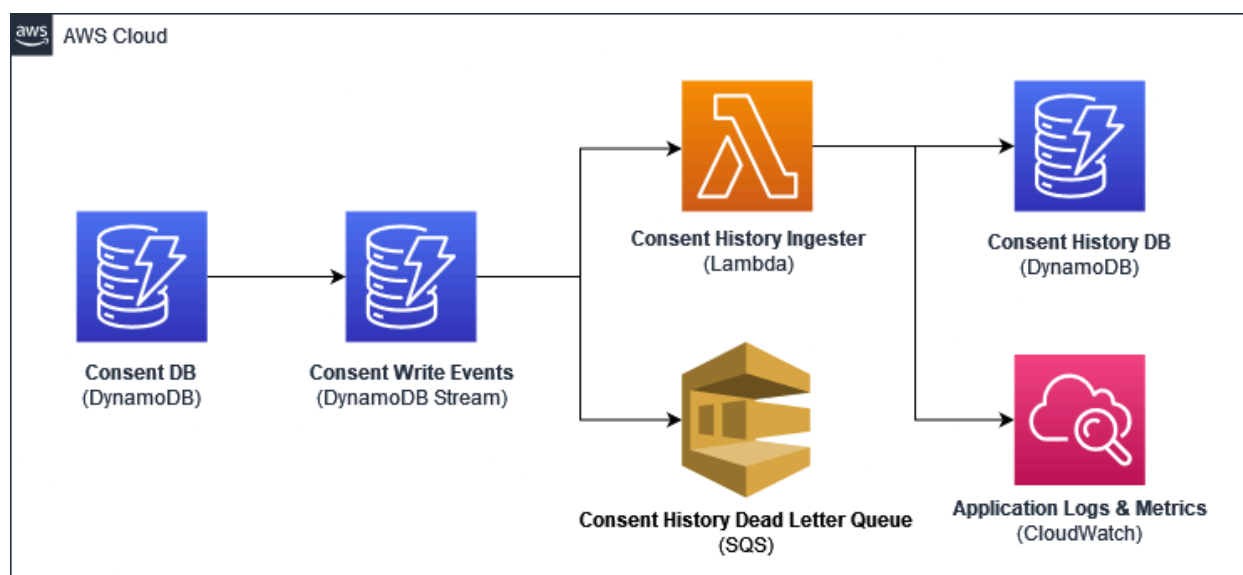
- Once approach 900+ requests/second during regular peak traffic, request quota increase and start migrating to EC2 on ECS, which is lower cost for continually running hosts that can handle higher-volume spikes without throttling, given a higher baseline of provisioned hosts and CPU + schedule-based auto-scaling policies.
- API interface: AWS REST API Gateway
 - Provides native support for client-specific metrics + rate limiting + throttling, authentication and authorization with resource-level access controls, API-gateway-level caching, SDK generation, execution logs, API method-level metrics, and request validation
 - Saved developer time is well worth the cost for up to 5M requests/month. Once reach that scale, revisit whether worth investing developer time to build functionality provided by REST API Gateway in the application layer behind a less expensive HTTP API Gateway or Application Load Balancer
- Firewall: AWS Web Application Firewall (WAF)
 - Configurable application firewall with available prebuilt rules for blocking common forms of malicious traffic, essential for protecting Internet-facing services

Consent history

Consent history ingestion architecture diagram

Consent DB updates will be automatically synced via DynamoDB Streams to an Consent History Worker Lambda Function, with an SQS dead letter queue catching any messages that the Lambda fails to process.

The Consent History Worker will write consent history items to a Consent History DynamoDB table, with application logs and metrics being emitted to CloudWatch.



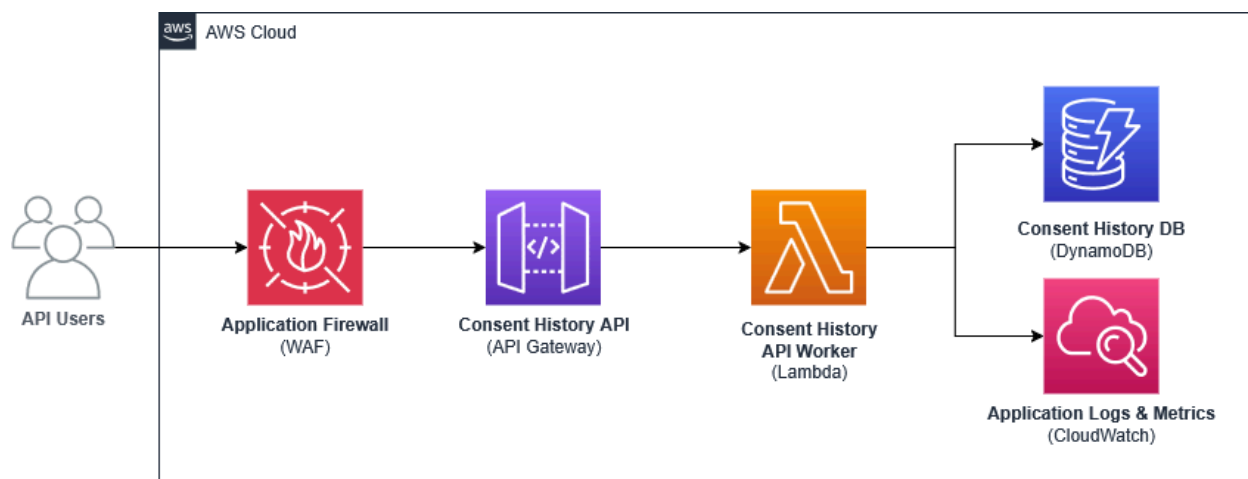
Technologies

- Consent History database: DynamoDB
 - Good fit for use cases, lower-cost and higher-availability than SQL-based databases, more effective than S3 for sub-second queries against multiple dimensions
- Compute service for processing consent update events: AWS Lambda
 - Simplest option to set up, and free for our expected scale
- Mechanism for streaming events from source Consent DB: DynamoDB Streams
 - Simpler to set up than Kinesis, and free for our expected scale

Consent history API architecture

The Consent History API architecture will match the Consent Management API, with its own API Gateway endpoints, API Lambda function, and Consent History DynamoDB table.

Consent History APIs will make read-only queries against the Consent History DynamoDB table.



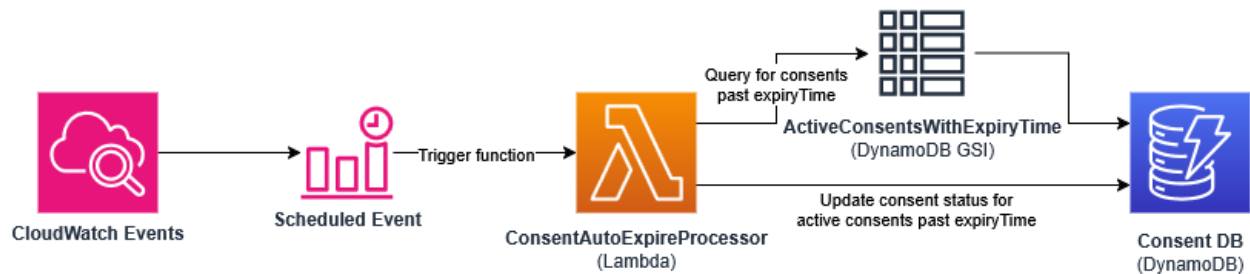
Consent auto-expiration

Consent auto-expiration architecture diagram

We will define a CloudWatch Event Rule with a scheduled event to periodically invoke a Consent Auto Expiry Processor Lambda function, which will query for all active consents that are past their expiryTime and update their status to expired.

To enable efficient look-ups of consents to auto-expire, we will set up a sparse DynamoDB GSI on the Consent table that tracks consents with non-null expiry times, using expiryHour as the GSI partition key and expiryTimeId (expiryTime + "|" + id) as the GSI sort key to automatically sort query results from oldest to newest expiry time while maintaining a unique key per record.

The Consent Auto Expiry Processor will query this GSI for each expiry hour from the past few days, and for any matching consents past their expiryTime, update their status to expired and remove them from the GSI.



Technologies

- **Scheduling: CloudWatch Events**
 - Good fit for use cases, simplest way to trigger a Lambda function on a periodic basis if don't need more advanced event bus or error handling functionality.
- **Compute: Lambda**
 - Simplest and lowest cost while have sparse expiryTime data, can migrate to higher cost, longer-lived compute types when needed.
- **Storage: DynamoDB**
 - Add an ActiveConsentsWithExpiryTime GSI tracking Consent partition key and expiryTime to the ServiceUserConsent DynamoDB table, to efficiently look up consents to auto-expire.
 - Submit DynamoDB UpdateItem requests to the ServiceUserConsent DynamoDB table for consents that have identified should be updated to an expired status.

API specifications and data models

Consent management API spec

<https://consent-management-platform.github.io/consent-management-api-models/v1/docs.html>

Consent data model

```
{
  // Consent ID attributes
  "serviceId": "string",
  "userId": "string",
  "consentId": "string",
  // Required consent attributes
  "consentVersion": "integer",
  "status": "string (enum values: ACTIVE, EXPIRED, REVOKED)",
  // Optional consent attributes
```

```
"consentType": "string",
"consentData": "object (up to 32 string key-value pairs)",
"expiryTime": "string (ISO 8601 datetime)"
}
```

Identifier attributes

The Consent data model will require that all consents have `serviceId`, `userId`, and `consentId` values, in order to support multiple types of consent between a user and service, and different user consents for different services.

Required non-identifier attributes

An integer `consentVersion` attribute will be used to guard against race conditions and overwriting consents with stale data, with update requests that do not increment the currently stored version rejected with a 409 Conflict error. A `status` attribute will store whether a given consent is active, expired, or revoked.

Optional attributes

A `consentType` string value will enable searches by consent type, and a `consentData` object will support flexible string key-value data to enable application-specific data customization. An `expiryTime` value can be set to automatically expire a consent at a given time.

Consent database schema

Query use cases:

- P0: Return consents associated with a { `serviceId`, `userId` } tuple
- P0: Create or update a specific consent with a unique { `serviceId`, `userId`, `consentId` } tuple value
- P1: Return consents associated with a { `serviceId`, `userId` } tuple filtered by `consentStatus`
- P1: Return consents associated with a { `serviceId`, `consentType` } tuple

Main `ServiceUserConsent` DynamoDB table:

- Partition key: `id <string>`, values of format `"{ServiceId}|{UserId}|{ConsentId}"`
- No sort key needed
- Direct queries against the table support `CreateServiceUserConsent` (P0), `UpdateServiceUserConsent` (P0), and `RetrieveServiceUserConsent` (P1, mostly for testing) APIs
 - Supports all consent write operations

`ConsentsByServiceUser` Global Secondary Index (GSI):

- From business logic point of view, something like ServiceId for the partition key and UserId for the sort key would make sense, however, this will likely have hot partition key issues if have one service with a massive number of consents
- Flipping to UserId for the partition key and ServiceId for the sort key will likely be more evenly distributed
- Partition key: `userId <string>`
- Sort key: `serviceId <string>`
- Projection type: ALL
- Queries against this GSI will support the ListServiceUserConsents API (P0), where can optionally filter for matching consents with a specific `ConsentType` value

ActiveConsentsWithExpiryTime Global Secondary Index (GSI):

- Partition key: `expiryHour`, values of format `"2010-12-03T11:00Z"`
- Sort key: `expiryTimeId`, values of format `expiryTime + "|" + the consent's id value` as the GSI sort key.
- Projection type: INCLUDE - include `consentVersion` as only non-key attribute in GSI
 - Only need the consent partition key, `consentVersion` and `expiryTime`
 - Scan the GSI in ascending order of `expiryTime` (oldest to newest)
 - Use consent partition key and `consentVersion` to submit UpdateItem DDB calls (`consentVersion` required for conditional writes), setting `consentStatus = "EXPIRED"`
 - Minimizing the GSI will improve query latency and lower storage and query costs

[P1, may not be needed for v1 launch] ConsentsByServiceAndConsentType (GSI):

- Partition key: `serviceIdConsentType <string>`, values of format `"{ServiceId}|{ConsentType}"`
- Sort key: `userId <string>`
- Projection type: INCLUDE - select specific fields to include in the GSI
 - Include `consentId`, `consentStatus`, and `expiryTime`
- Queries against this GSI will support companies and end customers searching for consents of a specific type associated with a specific service
- Will not include in beta launch, only add once have user feedback that existing list/search API is not sufficient for their use cases and validated that clients will use this service-level query functionality

Notes on choosing partition keys:

- *When storing data in DynamoDB, choosing a table partition key that is both unique across records and well-distributed across the key space increases table scalability and mitigates throttling issues around "hot partition keys" where most writes are clustered in a specific DynamoDB partition.*
- *Global Secondary Indexes (GSIs) should also be designed to distribute their queries across their partition key space to mitigate throttling issues on the GSI.*

Consent history API spec

<https://consent-management-platform.github.io/consent-history-api-models/v1/docs.html>

Consent history data model

```
{
  // (Partition key) Source consent record partition key
  "id": "string",
  // (Sort key) Unique event identifier
  "eventId": "string",
  // Before snapshot data, null for creation events
  "oldImage": "json",
  // After snapshot data, null for deletion events
  "newImage": "json",
  // Additional event metadata
  "eventType": "string (enum values: INSERT, MODIFY, REMOVE)",
  "eventTime": "string (ISO 8601 datetime with second granularity)"
}
```

Consent History data schema

ServiceUserConsentHistory DynamoDB table:

- Partition key: id
- Sort key: eventId

We use eventId instead of eventTime for the sort key, due to eventId being the only unique identifier for an event, while the same eventTime value may be duplicated across multiple events if there are multiple consent updates sent to the stream within the same second. The combined partition key + sort key for a DynamoDB entry must uniquely identify a given consent history item.

We use the compound consent id attribute as the partition key to enable efficient queries for the audit history of any given consent item.

This will enable querying the consent history table with consistentRead = true for any given consent item to retrieve all associated consent updates, as well as future GSIs and GSI queries against other consent dimensions.

Can add GSIs after obtaining more information on customer use cases.

Components

Consent history ingestion

The Consent DB table will have DynamoDB Streams enabled with the Stream View Option of `NEW_AND_OLD_IMAGES` to enable streaming the latest data from each consent update.

The following data will be parsed from each DynamoDB Stream event:

- `dynamodb.ApproximateCreationDateTime` - epoch timestamp representing approximate time of the consent update, rounded to the nearest second
- `dynamodb.Keys` - DynamoDB partition key and sort key of the source DB item
- `dynamodb.OldImage` - Prior content of source DB item after an update, or null if the event was the initial DynamoDB record creation
- `dynamodb.NewImage` - New content of source DB item after update
- `eventID` - unique identifier for the event
- `eventName` - INSERT, MODIFY, or DELETE

The DynamoDB Stream will configure the Consent History Lambda function as its target, where the Lambda function will parse stream events, convert them to the Consent History data model, and make DynamoDB PutItem requests to push the data to the Consent History DB table.

Ref:

https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_streams_StreamRecord.html

Consent auto-expiration

Consent Management API clients can optionally set an `expiryTime` timestamp on consent records.

Consent records where `status` = "ACTIVE" and `expiryTime` is non-null should be automatically updated to have `status` = "EXPIRED" if `expiryTime` is in the past, within a 24-hour SLA.

To support efficiently looking up consents that should be auto-expired, we will add an *ActiveConsentsWithExpiryTime* GSI to the consents table with `expiryHour` as the GSI partition key, and the `expiryTime` + "|" + the consent's `id` value as the GSI sort key.

This will automatically sort all query results in ascending order of expiry time (oldest to newest).

Consent Management API create/update requests should be updated to set `expiryHour` and `expiryTimeId` values based on `expiryTime` when the consent is active and has a non-null `expiryTime`, and otherwise set both values to null.

To auto-expire consents, we will set up a *ConsentAutoExpireProcessor* Lambda function that runs on an hourly schedule, where each function call runs the following algorithm:

1. Query the *ServiceUserConsent* DynamoDB table's *ActiveConsentsWithExpiryTime* GSI for each expiryHour from the last few days (will use 3 days for v1 launch), processing a single page of results at a time.
2. For each result from prior hours:
 - a. Submit a DynamoDB UpdateItem API request with id = { the current record's id }, and consentStatus = "EXPIRED"
3. For each result from the current hour's expiryHour bucket:
 - a. If expiryTime is in the past, submit a DynamoDB UpdateItem API request with id = { the current record's id }, and consentStatus = "EXPIRED"
 - b. Else if expiryTime is in the future, short-circuit and return with a success status

We will use individual synchronous DynamoDB UpdateItem API requests instead of submitting multiple concurrent asynchronous UpdateItem API requests for our v1 launch, to simplify implementation and maintenance while synchronous calls are sufficient to keep up with the scale of expiring consents.

We will use CloudWatch Events with an hourly cron schedule to trigger the Lambda function once per hour.

We will set up CloudWatch metrics and alarms to monitor for when:

- The *ConsentAutoExpireProcessor* Lambda function fails, or has not completed a job run within the past 2 hours.
 - This will be used to catch any system errors, unexpected edge cases, or timeouts that may need investigation and that did not self-resolve on subsequent job runs.
- The *ConsentAutoExpireProcessor* Lambda function duration was over 90% of the max Lambda runtime (15 minutes * 0.9 = 13.5 minutes) for 3 hourly job runs in a row.
 - This alarm indicates that we're approaching no longer being able to keep up with expiring consents with a 15 minute job runtime, and that we should implement optimizations, and/or migrate to a longer-lived compute type such as ECS, and/or consider running multiple concurrent workers that process different buckets of expiryTime periods.
- The oldest expiryTime in a given job run is over 6 hours in the past.
 - This indicates that for whatever reason, we have not been able to keep up with expired consents for 6 scheduled job runs and need to investigate.
 - This will be a fall-back alarm to catch a wider set of platform failures/gaps than other alarms, eg. if the Lambda function stops being triggered, or fails to start up, or fails to auto-expire consents for any reason.
 - This will not catch any bugs/failures that prevent the Lambda from running or getting to the point of emitting this metric, so still need the "lack of completed job runs" alarm listed above

- Can optionally simplify and save API calls by only emitting a metric for the first expired consent per job run, which will be the oldest expiryTime as of the start of the job run.

Consent API security mechanisms

All REST API Gateways will leverage the following security mechanisms:

1. [P0] Use [edge-optimized API endpoints](#), which leverage CloudFront to improve availability and reduce customer-facing latency to reach the API Gateway from anywhere around the world, and give access to basic CloudFront-provided DDOS mitigations in exchange for a slightly higher cost compared to regional API endpoints.
2. [P0] Enforce request model validation in the API Gateway layer, to automatically reject requests that do not match expected patterns for request attributes.
3. [P0] Enforce API Gateway resource policies that only allow requests from known authorized users, such as:
 - a. Specific IAM roles
 - b. Specific customer AWS IAM roles or AWS accounts
 - c. Specific customer IP ranges, if need to support customers whose services are deployed outside of AWS
4. [P0] Enforce throttling limits on APIs, with very low default limits to mitigate the impact of DDOS attacks on backend services.
5. [P1] Enforce that all API calls provide API keys, with customer-specific usage plans that have throttling limits matching their required API usage.
 - a. These are static and typically difficult to rotate, and passed in plaintext HTTP request headers, so our system should not rely on them never becoming public.
 - b. We should implement a Lambda Authorizer to enforce authentication/authorization, and can potentially add logic to validate the auth token is allowed to use the provided API key.
6. [P1] Implement a Lambda Authorizer that enables user-specific authentication/authorization logic, dynamic access decisions, token revocation, and passing user identities to the backend.

The CDK code and Lambda Authorizer code should support client config being retrieved from some dynamic source outside of source control, so that we can deploy different client config for each deployed AWS account, and keep client AWS account IDs and IP ranges secret.

API client config injection

We want to maintain open-source service and infrastructure code repositories that are common to all deployed instances of the Consent Platform, while allowing for Business-Client-specific deployments which have different allow-listed API clients for each client-specific AWS account.

In order to keep client-specific API access configurations confidential, and allow for different allow-listed API client AWS account IDs and/or IP ranges, we will store Business Client API configuration settings in a centralized client config AWS account within AWS AppConfig.

Each Business Client's API configuration settings will be stored in a separate AppConfig configuration, with an AppConfig application > AppConfig environment > AppConfig configuration path specific to that Business Client + environment (eg. BusinessA Prod), and the given client's associated deployment pipeline will copy the client's API configuration files to a standard file path expected by the CDK code.

The CDK code will then check for the existence of custom API client config at that standard file path, and if it exists, parse the JSON file into specific configuration settings to apply to:

1. [P0] API Gateway resource policies
2. [P0] API Gateway usage plans which will have client-specific access keys and throttling settings
3. [P1] API Gateway Lambda Authorizer environment variables which will be used to enforce fine grained resource access controls within

For our initial launch, we will implement access controls at the API Gateway resource policy layer, which allow us to restrict access to API methods to allow-listed AWS account IDs and IP ranges, while allowing those API clients to query any resources under that API.

If/when we have interested business clients who need fine grained access controls restricting some of their API clients to only access specific resource namespaces, then we can implement API Gateway Lambda Authorizers that support more advanced access control use cases.

Details at [Design for dynamic API client config in AWS](#)

Implementation

Test plan

Unit tests

Each code package will have in-package unit tests that are run whenever the package is built.

All package unit tests must pass before code will be deployed.

Integration tests

Each service will have integration tests that validate service-level expectations after invoking the deployed APIs or other workflows.

Eg. API integration tests will invoke the deployed API Gateway endpoints and validate each sample request results in the expected response.

All integration tests must pass before allowing code changes to continue to deploy to the subsequent stage.

For v1 launch, will run integration tests via GitHub Workflows + AWS CLI as the lowest effort mechanism to ensure all APIs work as expected after code deployments or when run on-demand.

Project plan

Milestone	Deliverable	Description	Estimated dev effort	Status
Milestone 1: Foundational consent operations	Consent Management API base operations	Consent Management API supports consent create, update, retrieve, and list operations Includes setting up: • API models - DONE • API service - DONE • API Gateway - DONE • API documentation, with automatic updates synced from API models - DONE	1 month	Completed 2024-07-01
	Consent history generation	All consent update operations are synced to Consent History DB Includes setting up: • Consent History DB - DONE • DynamoDB Stream from Consent DB to ingester - DONE • Ingester service code - DONE • Ingester Lambda infra - DONE • Events dead letter queue - DONE	1 week	Completed 2024-12-24
	Consent Management API integration tests	Consent Management API endpoints should all have integration tests that run as part of the deployment workflow.	1 week	Completed 2025-03-22
Milestone 2: Consent History API	Consent History API for specific consent, without pagination	Support queries for all consent updates for a given { serviceId, userId, consentId } tuple, without pagination Includes setting up: • API models - DONE • API documentation, with automatic updates synced from API models - DONE • API service - DONE • API Gateway - DONE	2 weeks	Completed 2025-04-05
	Consent History API for specific service user, without pagination	Support queries for all consent updates for a given { serviceId, userId } tuple, without pagination	1 week	Completed 2025-04-10

	Consent History integration tests	The Consent History workflow should have integration tests that run as part of the deployment workflow - GetHistoryForServiceUserConsent integ tests - DONE - GetHistoryForServiceUser - DONE	1 day	Completed 2025-04-10
Milestone 3: Consent auto-expiration support	Automatic consent expiry support	Consent records past their expiry time are automatically updated to "EXPIRED" status, without deleting them from the data store Includes setting up: - ActiveConsentsWithExpiryTime GSI - DONE - Consent Management API code changes to set/clear the GSI key based on status/expiryTime values - DONE - Skeleton ConsentAutoExpiryProcessor Lambda function deployed to AWS - DONE - ConsentAutoExpiryProcessor service code to scan the GSI and update matching records that are past their expiryTime - DONE - CloudWatch Event to trigger worker hourly - DONE - CloudWatch alarms/dashboards - DONE	2 weeks	Completed 2025-04-18
Milestone 4: Client config	Implement API client config	Write design for mechanism to inject client-specific API client config into single-tenant platforms. - DONE, 1 week effort Set up CDK package and AWS account for centralized client config account. Implement mechanism to inject API client config specific to a given business client deployment into the CDK code, while keeping business client configs confidential. Set up automatic validation of client config files against expected schema and attribute value patterns. - In progress Set up integration with API Gateway resource policies to allow-list specific client AWS account IDs and IP ranges.	TBD	In design
Milestone 5: Harden API Gateways	Enforce API usage plans	All API Gateways provide per-customer access keys with custom usage plans. All API Gateways enforce that customer-specific access keys be provided in API requests. [P1] Extend API client config to provide per-client API throttling limits.		
	Implement Lambda Authorizer	Implement a custom Lambda Authorizer for each API Gateway, which:		

		• [P0] Supports access by authorized IAM Roles • [P0] Supports access by authorized client AWS accounts • [P0] Supports access by authorized client IP ranges • [P0] Rejects requests that are not AWS-signed • [P0] Rejects requests that do not match a supported client • [P1] Implements request-path-based authorization logic to validate the client's access to the specific consent resources, eg. validate they own the services whose service user consents are being read/written ◦ Low priority if platform is single-tenanted with each customer having their own deployed AWS account.		
Milestone 6: Zero-trust encryption	Zero-trust encryption model	Implement zero-trust encryption model guaranteeing that all data provided by onboarded companies cannot be read by us, only by the client companies		
	Zero-trust encryption support built into API clients	Integrate zero-trust functionality into API clients to simplify onboarding		
Milestone 7: Operational readiness	Complete operational readiness tasks	• Operational and business metrics set up with consolidated operational dashboard • Alarms and runbooks set up for service/component unavailability, failure cases, and latencies breaching SLA • Alarms and runbooks set up for AWS services approaching default service limits/quotas • Resource access policies and per-client API metrics, rate limiting, and throttling set up • Integration tests, API canaries with associated availability alarms, and load tests set up • Continuous deployment pipeline set up with automated validations of code changes against a non-prod environment prior to any changes deploying to prod • Billing alarms configured with automatic notifications when breach expected thresholds • Prod account hardening, MFA required for account login, least-privilege access permissions set up for AWS accounts to enable maintenance work and on-call troubleshooting with minimal risk of destructive actions • Protect main branches and enforce successful builds + quality checks against a feature branch before merging to main • Web application firewall set up to protect Consent Management API and Consent Audit Trail API, with most relevant AWS-provided rule-sets configured and enabled		

Milestone 8: Beta release	Onboard first beta customers	• Complete user documentation with onboarding instructions, released API clients, and sample integration code • Onboard beta customers, monitor usage and collect feedback for prioritized backlog • Optimize service latency and hardware spending based on actual usage • Iterate on highest priority bug fixes and feature requests, determine priorities for next launch		
Milestone 9: Incremental feature improvements	Incremental feature releases based on user feedback	• Add pagination support for GetHistoryForServiceUserConsent API if expect enough updates per consent that will need multiple pages • Add API support for searching consents by { serviceId, consentType } and { serviceId, userId, consentType } • Add Consent list API support for filtering by record status • (...Other features from prioritized backlog)		

Project resources

GitHub organization: <https://github.com/Consent-Management-Platform>

- All service and infrastructure packages will be released under this GitHub organization.

Semgrep security code scan dashboard:

https://semgrep.dev/orgs/consent_management_platform

FAQ

Why are we not providing a consent management UI?

Each company will likely have their own specific requirements for how to surface consent management controls to their customers, and integrate these controls into existing websites and applications they manage.

Therefore we will focus our platform's initial launch on providing robust API services that these companies can integrate with to meet their backend consent management needs.

Where are the private VPC subnets in the design diagram?

No private VPC subnets are needed for our use cases, since both DynamoDB and Lambda are AWS-managed services that live in AWS-managed VPCs, not in our VPCs.

RE <https://docs.aws.amazon.com/lambda/latest/dg/configuration-vpc.html>:

"Every Lambda function runs inside a VPC that is owned and managed by the Lambda service. These VPCs are maintained automatically by Lambda and are not visible to customers. Configuring your function to access other AWS resources in an Amazon VPC has no effect on the Lambda-managed VPC your function runs inside."

We will use IAM resource policies to restrict access to our DynamoDB tables so that only our API Lambda Functions can query them, and restrict access to our Lambda Functions so that they can only be queried by our API Gateway, so there are no resources that need to live in private subnets.

If we were using relational RDS databases instead of DynamoDB, this would be different. Since RDS instances are hosted in VPCs we manage, we would want to host the RDS instances in private subnets to protect them, and then configure our Lambda Function to be "VPC-enabled" and grant it access to the VPC's private subnet so that it can access the RDS instance.

As discussed in <https://medium.com/lumigo/to-vpc-or-not-to-vpc-pros-and-cons-in-aws-lambda-aeaad000432> and <https://medium.com/@clemen.kok/should-your-lambda-be-in-a-private-subnet-c1bb77682315>, making Lambda Functions VPC-enabled adds complexity, costs, and scalability concerns that you need to monitor for, and isn't necessary or recommended unless you need to connect your Lambda functions to resources within protected VPCs.

Appendices

Appendix 1: Scale estimates

Summary

- Expect service to be able to start off provisioned to support 5 GB data and 5M transactions/month, with peak traffic of 1 write/second and 25 reads/second.
- Design service to be able to scale to support 2.5 TB data and 500M transactions/month, with peak traffic of 100 writes/second and 2.5k reads/second.

	Data size	Read transactions/month	Write transactions/month	Peak reads/second	Peak writes/second
100k end customers	25 GB	5M	100k	25	1
1M end customers	250 GB	50M	1M	250	10
10M end customers	2.5 TB	500M	10M	2.5k	100

Scale estimate calculations

- For baseline, assume have 1M end customers, can divide/multiply by resulting scale

estimates

- Data size:
 - Assume 1M end customers with average 5 KB consent data/customer -> ~5 GB consent data in table
 - Assume data has replication factor of 3 per DB server across table and indexes
 - Assume 1 master database with 2 replicas for availability -> 3 total DB servers
 - 5 GB consent data * 3 copies/server * 3 servers = 45 GB ~ 50 GB consent data
 - 4x audit history size -> 200 GB audit history
- Peak traffic from end customers:
 - Assume the average customer triggers 10 read requests/year, 10 write requests/year
 - 1M customers * (10 reads/year, 10 writes/year) -> 10M reads/year, 10M writes/year
 - Assume traffic is clustered with:
 - 20% of annual traffic clustered to a single month
 - 50% of that month's traffic clustered to a single week
 - 20% of that week's traffic clustered to a single day
 - 20% of that day's traffic clustered to a single hour
 - That hour's traffic uniformly distributed
 - Peak transactions/second is then (10M reads/year, 10M writes/year) * 0.2 yearly traffic/peak month * 0.5 peak month traffic/peak week * 0.2 peak week traffic/peak day * 0.2 peak day traffic/peak hour * 1 hour/3600 seconds
 - $10,000,000 * 0.2 * 0.5 * 0.2 * 0.2 / 3600 \approx 11$ reads/second, 11 writes/second
- Peak traffic from companies:
 - Assume the average company integrating their workflows with the consent platform has the following workflows:
 - Reading consent data for a specific {service, user} tuple on every relevant interactions that may use that data
 - Assume average of 100 such interactions/year/customer, with same peak clustering pattern as the above distribution of end customers managing their consent settings
 - 1M customers * 100 * 0.2 * 0.5 * 0.2 * 0.2 / 3600 $\approx$ peak traffic of 111 reads/second
 - Async backend workflows querying consent data after each customer interaction
 - Assume on average trigger one such query per customer interaction -> same peak of ~111 reads/second
 - Async daily jobs querying all consent data
 - Assume 1M queries/day uniformly spread across 6 hours
 - 1M queries/day * 1/24 days/hour * 1/3600 hours/second $\approx$ 11 reads/second
- Total peak traffic:
 - 11 writes/second $\approx$ 10 writes/second
 - 244 reads/second (11+111+111+11) $\approx$ 250 reads/second
- Average monthly transactions:
 - Writes: 1M customers * 10 writes/year * 1/12 years/month $\approx$ 1M writes/month
 - Reads: ~50M reads/month
 - End customers: 10M/12 $\approx$ 833k/month
 - Companies: $1M * 100 / 12 * 2 + 1M * 365 / 12 = 1M * (100 / 12 * 2 + 365 / 12) = 47M$ /month

Appendix 2: Design alternatives considered

Consent management database

Technical database requirements:

1. Support peak traffic of over 2.5k reads/second and 100 writes/second
2. Support storing 250 GB in a consent data table
3. Support storing 1 TB in an audit history data table
4. Support flexible consent data attributes such as JSON objects that may have different internal schemas depending on the company use case
5. Provide point-in-time data backup and recovery
6. p95 read latencies $\leq 20\text{ms}$ for retrieving {service, user} and {service, user, consent type} tuples
7. p95 write latencies $\leq 50\text{ms}$ for creating/updating consent records
8. Availability SLA $\geq 99.9\%$

	MySQL/PostgreSQL on RDS	Aurora RDS Serverless	DynamoDB
Maximum transactions/second	Dependent on instance type, larger instance types support higher IOPS. Can scale from 1k-256k IOPS.	AWS benchmarks using r3.8xlarge: 500k reads/second 100k writes/second Aurora benchmarks claim 5x higher performance than MySQL, 3x higher than PostgreSQL	No hard limit, can request increases to read/write quotas as needed.
Maximum connections	Mainly constrained by memory and impact on performance. Default typically set to 100, can increase to 100k for MySQL, ~8.4M for PostgreSQL, while very high limits can negatively impact performance.	7000	No limit
Maximum table size	MySQL: 16 TB PostgreSQL: 32 TB	64 TB	No limit
Maximum record size	MySQL on InnoDB engine: • Hard limit of ~64 KB not including TEXT/BLOB fields stored separately, other than their 9-12 byte pointers • Constrained to under half	Based on choice of MySQL or PostgreSQL backend	400 KB

	of InnoDB page size for InnoDB page sizes of up to 32 KB, around 16 KB for max InnoDB page size of 64 KB PostgreSQL: 4 GB		
Maximum number columns	MySQL on InnoDB engine: 1017 PostgreSQL: 250-1600 depending on column types	Based on choice of MySQL or PostgreSQL backend	No limit
Maximum query page size	1 GB	1 GB	1 MB
Maximum number of indices	MySQL: 64 PostgreSQL: No limit	Based on choice of MySQL or PostgreSQL backend	20 GSIs per table 5 LSIs per table
JSON attribute support	Yes	Yes	Yes
Point-in-time recovery support	Yes	Yes	Yes
Availability SLA	99.95%	99.99%	99.999% for global tables 99.99% for standard tables
Maintenance effort	Medium	Low	Low
Expected hardware cost to support 1M customers	\$5.8k/month for 1 master with 2 replicas	\$1.15k+/month for 1 master with 2 replicas	\$130/month
Expected hardware cost to support 10M customers	\$33.5k/month for 1 master with 2 replicas	\$11.5k+/month for 1 master with 2 replicas	\$1.3k/month

Recommendation: DynamoDB

- Good fit to expected use cases which align with simple key-value queries
- Highly scalable with no hard concurrency or throughput limits, minimal overhead to increase limits
- Lowest expected hardware costs by far, especially as number of users increases

Notes on DynamoDB scaling:

- There is no hard limit on read/write throughput, while accounts have default quotas that can be increased upon request
- Default quotas per table (can request increases):
 - 40k read capacity units (RCUs) per second
 - 40k write capacity units (WCUs) per second
- 1 read capacity unit = 1 strongly consistent read, or 2 eventually consistent reads, per second for items <= 4KB
- 1 write capacity unit = 1 write/second for items <= 1KB

References:

- Aurora scaling:
<https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/AuroraMySQL.Managing.Performance.html>
- RDS quotas and constraints:
https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Limits.html
- RDS pricing:
 - <https://aws.amazon.com/rds/mysql/pricing>
 - <https://aws.amazon.com/rds/postgresql/pricing>
- MySQL InnoDB storage engine constraints:
<https://dev.mysql.com/doc/refman/8.0/en/innodb-limits.html>
- PostgreSQL constraints:
 - <https://www.postgresql.org/docs/current/limits.html>
 - https://wiki.postgresql.org/wiki/FAQ#What_is_the_maximum_size_for_a_row.2C_a_table.2C_and_a_database.3F
- DynamoDB quotas and constraints:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ServiceQuotas.html>

Consent management API compute service

Technical database requirements:

1. Support peak traffic of over 2.5k reads/second and 100 writes/second
2. p95% read latencies <= 30ms for retrieving {service, user} and {service, user, consent type} tuples
3. p95% write latencies <= 50ms for creating/updating consent records
4. Availability SLA >= 99.9%

	Lambda	Fargate on ECS	EC2 on ECS
Maximum transactions/second	No limit	No limit	No limit
Maximum concurrency	No hard limit Default limit of 1k concurrent Lambda executions/account but can request quota increase	No hard limit Constrained by default limit of 5k tasks/service but can request quota increase	No hard limit Constrained by default limit of 5k tasks/service but can request quota increase
Maximum execution time	15 minutes	N/A	N/A
Who manages infrastructure	AWS	AWS	You
Who manages auto-scaling	AWS	AWS, while you can add scaling config if you want	You
Cold start latency	1-2 seconds if cold start, but can minimize by periodically pinging service to “keep warm”	N/A unless provisioning new task from auto-scaling	N/A unless provisioning new task from auto-scaling

How handles sharp traffic spikes	If sharply spike above 1k calls/second, can take a few minutes to provision resources to handle higher traffic, throttles calls above 1k/second. Can mitigate by provisioning resources if expect spikes above that scale, however, then always paying for that higher provisioned usage threshold, compared to only paying for actual usage.	If sharply spike above what current hosts can handle, throttles/drops traffic above capacity until auto-scales to adjust based on your auto-scaling criteria, autoscaling can take a couple minutes to provision new tasks. Can mitigate by provisioning fleet with a minimum baseline to handle regular spikes.	If sharply spike above what current hosts can handle, throttles/drops traffic above capacity until auto-scales to adjust based on your auto-scaling criteria, autoscaling can take a couple minutes to provision new tasks. Can mitigate by provisioning fleet with a minimum baseline to handle regular spikes.
Optimal use cases	Short-lived stateless applications with < 1k requests/second Sporadic traffic where not using most capacity for majority of day	Long-lived tasks that stop running once complete Long-lived applications where willing to pay slight premium to offload infrastructure and auto-scaling maintenance to AWS	Services with continuous traffic and predictable resource capacity requirements, where right-sizing instance types can save money over Fargate
Availability SLA	99.95%	99.99%	99.99%
Maintenance effort	Low	Low-medium	Medium
Expected hardware cost to support 1M customers	\$22.75	\$96.42	\$33.60
Expected hardware cost to support 10M customers	\$11.1k/month if need 2k provisioned capacity to handle traffic spikes without throttling \$229.30 if can accept throttling for up to a few minutes for spikes in traffic above 1k/second	\$835.62/month	\$336/month

Recommendation:

- Start with AWS Lambda while have < 3M end customers on the same platform
 - Set up "canary" workers that periodically making read requests to keep service warm
 - Lowest-maintenance, lowest-cost solution up until see traffic closer to 1k requests/second and need to provision long-term tasks to handle spikes
 - Expecting to support our scaling needs up to ~3.8M end customers (10/2.6 given expect 2.6k requests/second at peak for 10M customers)
- When onboarding companies large enough that we anticipate > 1k requests/second, migrate to EC2 on ECS to lowest cost for higher-scale API service, using reserved instances and auto-scaling by both CPU utilization and a schedule based on time of day and day of week

Availability SLAs:

- Lambda: <https://aws.amazon.com/lambda/sla/>
- ECS: <https://aws.amazon.com/ecs/sla/>

Consent management API interface

	REST API Gateway	HTTP API Gateway	Application Load Balancer
Native integration with Lambda	Yes	Yes with limitations No support for routing based on request path parameters or request body attributes, only simple pass-through to Lambda	Yes
Native integration with WAF	Yes	No	Yes
Supports resource access policies	Yes	No	No
Supports per-client usage metrics + rate-limiting + throttling	Yes	No	No
Set-up effort including work to set up WAF, resource access policies, and client-level metrics + rate limiting + throttling	Low	High	Medium
Availability SLA	99.95%	99.95%	99.99% for multi-AZ load balancer 99.9% for single-AZ load balancer
Expected hardware cost to support 1M customers	\$180/month without caching \$210/month with caching	\$51/month	\$30/month
Expected hardware cost to support 10M customers	\$1.7k/month without caching \$1.8k/month with caching	\$490/month	\$80/month

Recommendation:

- Start with REST API Gateway for rich feature set with minimal set-up effort, for reasonable cost up to 1M end customers
- Once reach 1M end customers or > 5M API requests/month, whichever comes first, revisit whether worth the hardware savings at higher traffic volumes to invest developer time into building functionality provided by REST API Gateway in the application layer on top of an HTTP API Gateway or Application Load Balancer

Main options under consideration:

- REST API Gateway
 - First choice for natively supported functionality including built-in WAF integration, resource access policies, per-client usage metrics + rate-limiting + throttling, caching, request body validation, and AWS X-Ray tracing
 - Feature set we would use for this platform would likely otherwise take over a month of developer time to implement
 - Reasonable price up to the 1M customer scenario, 10M customer scenario becomes more expensive, still potentially worth it for reduced development time if have sufficient funding at that scale of customers for an additional \$1.5k/month in costs for supporting 10M customers to be worth it.
- HTTP API Gateway
 - Prefer REST API Gateway for additional functionality, and if were extremely price sensitive, would choose application load balancer instead
- Application load balancer
 - Strong candidate if need to support hundreds of millions of requests/month for under \$100/month, however, lacks major features provided by REST API Gateway such as resource access policies and per-client usage metrics + rate-limiting + and throttling which would prioritize if hosting a multi-tenancy platform
- Network load balancer
 - Ruling out since lacks significant application-layer functionality we would want to be handled at the API interface layer such as HTTP layer routing, access controls, WAF integration, and per-client rate limiting and throttling

API Gateway options:

- REST vs HTTP API Gateway: Prefer REST if can afford to pay cost
 - Only REST API Gateways support:
 - Built-in AWS WAF integration
 - Edge-optimized and private endpoints
 - Resource access policies
 - Per-client rate-limiting and throttling
 - Caching
 - Request body validation
 - Execution logs and X-Ray tracing
 - Only HTTP API Gateways support:
 - Private integrations with Application Load Balancers (not needed for our use case)
 - REST API Gateways have a higher price in exchange for the extended functionality
 - Both provide 1M free requests/month
 - After first 1M free requests:
 - HTTP API Gateways charge \$1/M requests for next 300M requests

- REST API Gateways charge \$3.5/M requests for next 333M requests
 - If can afford it, prefer REST API Gateways for wider feature set which would plan to use
- Edge-optimized vs regional: **Edge-optimized** (if using REST API Gateway)
 - Edge-optimized gateways are only available for REST API Gateways
 - Edge-optimized gateways automatically set up CloudFront resources in multiple global regions to provide low-latency client access globally
 - Regional gateways may provide slightly lower latency if all clients are hosted in the same AWS region - cannot guarantee this for our use case
- Public vs private: **Public**
 - Public APIs are the easiest to set up and have higher availability of 99.95%, can protect via WAF rules and API Gateway authentication/authorization
 - Private APIs require additional infrastructure with AWS Private Link and service owner defined routing configuration, and have a lower availability of 99.9%
 - Recommend public API for higher availability and simpler security configuration that is easier to get right while still protecting the service

Load balancer options:

- Application load balancer
 - Has access to application layer of packets, supports application-level traffic manipulation and request content-based routing, SSL termination, and session persistence
 - Supports HTTP, HTTPS, and gRPC protocols
 - Provides native integrations with AWS services such as Lambda and API Gateway that reduce developer set-up effort
 - Supports mutual TLS for validating clients are who they say they are, and only allow approved clients to connect
 - Would leverage this functionality to offload authentication to the load balancer rather than need to implement similar logic in application code if using a load balancer instead of an API Gateway
- Network load balancer
 - Only has access to up to the network layer of packets, simpler routing based on IPs and server availability, highly reliable and low-latency
 - Supports TCP, UDP, and TLS protocols
 - Lowest possible cost for routing traffic to backend API workers
 - Requires authentication and authorization logic to be handle in backend application code
- If load balancer is receiving HTTP requests (eg. API interface layer for REST API clients) and want flexibility to route traffic based on HTTP-layer content (eg. request path parameters), should use an Application Load Balancer
- Prefer application balancer over network load balancer for API services that don't need to handle millions of requests per second, for lower developer effort to integrate with

other services, flexible HTTP-based routing, and basic authentication if not using API Gateway

References:

- HTTP vs REST API Gateways: <https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-vs-rest.html>
- Edge-optimized, regional, and private API Gateways: <https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-endpoint-types.html>
- API Gateway pricing: <https://aws.amazon.com/api-gateway/pricing/>

Consent history database

Technical database requirements:

1. Support subsecond queries for all consent updates associated with a given `serviceld`, or a given `userId`, or a given { `serviceld`, `userId` } pair
2. Support peak traffic of 100 reads/second and 100 writes/second
3. Support storage of 1 TB+
4. Support flexible consent data attributes such as JSON objects that may have different internal schemas depending on the company use case
5. Provide point-in-time data backup and recovery
6. p95 read latencies $\leq 50\text{ms}$
7. p95 write latencies $\leq 50\text{ms}$
8. Availability SLA $\geq 99.9\%$

Recommendation: DynamoDB

- DynamoDB is the least expensive, highest availability major database option for our expected data scale that supports efficient queries across multiple dimensions such as `serviceld`, `userId`, or { `userId`, `serviceld` } pairs
- See above “Consent management database” design alternatives and tradeoffs table comparing RDS options and DynamoDB
- Ruling out non-database storage options such as S3 which can be less expensive for large data storage but are more cumbersome to set up to efficiently query across multiple dimensions

Consent history ingestion workflow

Technical requirements:

1. [Data freshness] Consent write operations must be reflected in the consent history within a p95 latency of 5 seconds
 - a. It's acceptable for the consent history to have some latency in receiving updates, but should reflect all updates within a few seconds so that downstream customers can refresh a company's consent history UI to observe their changes
2. [Fault tolerance] Consent write operations must be eventually propagated to the consent history even if a given system component is temporarily offline

3. Consent history entries must never be deleted or overwritten, only added

	DynamoDB Stream and Lambda with SQS DLQ	Kinesis Data Stream and Lambda with SQS DLQ	Consent Management API Writing Consent History DB	Consent Management API Invoking Consent History Lambda	Consent Management API Pushing to Consent History SQS
Event deduplication	Yes	No Kinesis Data Stream can have duplicate events	Yes	Yes	Yes if use FIFO SQS queue
Unprocessed event data retention	24 hours in DynamoDB Stream, 14 days in SQS DLQ	1-7 days in Kinesis, 14 days in SQS DLQ	None if Consent Management API fails before writing to Audit Trail	None if Consent Management API worker fails before invoking Audit Trail Lambda function	None if Consent Management API worker fails before writing to SQS
Fault tolerance	Can replay from DynamoDB within 24 hours, replay from SQS DLQ within 14 days	Can replay from Kinesis within configured retention period (1-7 days), replay from SQS DLQ within 14 days	None	None if Consent Management API worker fails before invoking Audit Trail Lambda function Can replay from Audit Trail DLQ within 14 days if failure caught within Audit Trail Lambda invocation	None if Consent Management API worker fails before writing to SQS Can replay from SQS within 14 days if successfully push SQS message
Concurrency limits	2 concurrent consumers per DynamoDB Stream shard No hard throughput limits, default DynamoDB and Lambda throughput limits but can request quota increases	20 concurrent consumers per Kinesis Data Stream shard No hard throughput limits, default Lambda throughput limits but can request quota increases	N/A	No hard limit Default limit of 1k concurrent Lambda executions/account but can request quota increase	Max 20k in-flight messages for FIFO queues
Set-up and maintenance effort, including event deduplication logic	Medium-Low	Medium-High	Low	Medium-Low	Medium
Expected hardware cost for 1M customers	\$0.84/month	\$293.63/month			
Expected hardware cost for 10M customers	\$11.64/month	\$311.51/month			

Recommendation: Start off with DynamoDB Stream and Lambda with DLQ

- Data streams from the Consent DB to the Consent History DB are more fault tolerant than having the Consent Management create/update APIs pushing events.
- AWS hardware costs for DynamoDB Streams pushing consent updates to an Consent History Lambda Function with a DLQ are minimal for our use cases, even up to 10M end customers.

- Set-up effort is lower with DynamoDB Streams compared to Kinesis Data Streams.
- Do not expect concurrency limits of DynamoDB Streams to become a bottleneck in the near future, can consider migrating to Kinesis Data Streams with minimal redesign needed if throttling becomes an issue as scale up.
 - Clustering of consumers within common DynamoDB shards is based on how distributed the underlying Consent DB data is across DynamoDB partitions, which should be well-distributed based on choices of partition keys.
 - Even in the case where there are many consent updates sharing the same source Consent DB DynamoDB partition, 2 concurrent consumers per corresponding DynamoDB Stream shard * 1 processed event/0.03 seconds assuming average 30ms latency $\approx$ 66.67 requests processed/second/shard, and at peak we only expect a total of 100 consent updates/second which will be distributed across many shards.
 - May just need to throttle consent updates from automated client integ tests.

Comparison of DynamoDB Streams and Kinesis Data Streams:

<https://aws.amazon.com/blogs/database/choose-the-right-change-data-capture-strategy-for-your-amazon-dynamodb-applications/>

DynamoDB Stream limits:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ServiceQuotas.html#limits-dynamodb-streams>

FIFO SQS queue limits:

<https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/quotas-fifo.html>

Option 1: DynamoDB Stream and Lambda with SQS Dead Letter Queue

DynamoDB has native support for streaming all table item updates to an AWS Lambda function where we can apply any processing logic and write consent history items to our Consent History DynamoDB table.

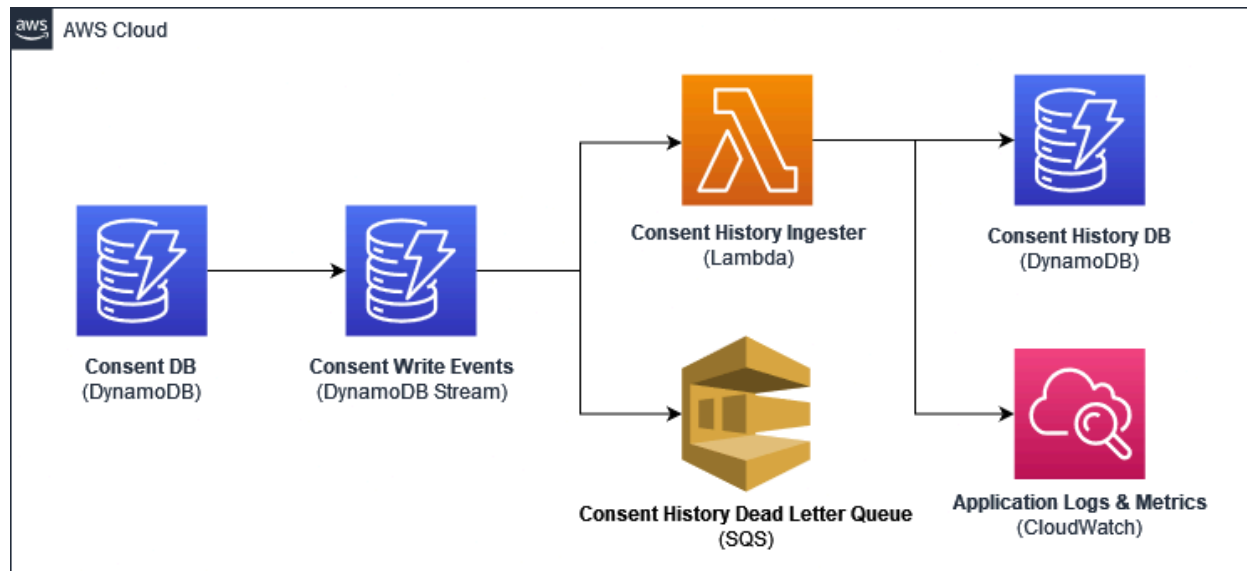
Lambda is the most cost-effective compute type for our expected traffic pattern of sparse, short-lived executions, with no more than 100 write requests/second.

With DynamoDB Streams, we automatically get event deduplication and first-in-first-out event ordering.

However, DynamoDB Streams have a data retention period of only 24 hours, so if our Consent History workflow is broken for over a day, we will be unable to replay the events from that time period. Therefore, for fault tolerance, we will want to configure an SQS “dead letter queue” (DLQ) to capture any events the Lambda fails to process, which increases our effective data retention period to the 14 days supported by SQS.

Lambda's native DLQ option will not support our use cases, as it may not capture failed requests from async event sources, however, we can configure a DLQ in the event source mapping that connects the DynamoDB Stream to the Lambda function.

As per <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html>, "If you perform a PutItem or UpdateItem operation that does not change any data in an item, DynamoDB Streams does not write a stream record for that operation." This is consistent with our use case where we only want consent data changes to result in Consent History items.



Ref for how to set up dead letter queues for event source mappings:

<https://aws.amazon.com/blogs/compute/new-aws-lambda-controls-for-stream-processing-and-a-synchronous-invocations/>

Option 2: Kinesis Stream and Lambda with SQS dead letter queue

Kinesis Data Streams are a general-purpose, highly scalable AWS-managed service for event streaming, which we could use in place of DynamoDB Streams to sync Consent DynamoDB table item updates to a Lambda function to process, with an event source mapping dead letter queue catching events that failed processing.

Kinesis has advantages of:

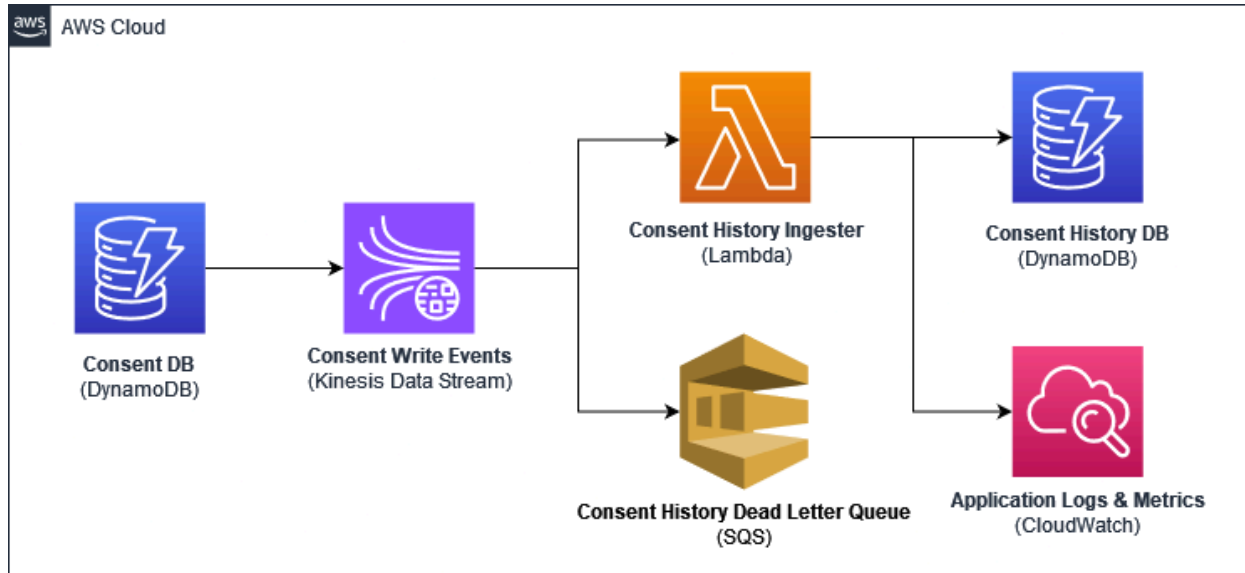
- Data retention of up to 7 days compared to 1 day with DynamoDB Streams
- Higher max consumers than DynamoDB Streams per shard with enhanced fan-out
- No throughput limits

Downsides of using Kinesis:

- Events may be duplicated
- Events may be received out-of-order
- Higher price than DynamoDB Streams

- Higher set-up and maintenance effort compared to DynamoDB Streams

Kinesis is a flexible option for data streaming that may be preferable if DynamoDB Streams does not meet our scaling requirements, otherwise would prefer DynamoDB Streams for lower cost, simpler set-up, and built-in event deduplication and ordering.



Option 3: Updating the Consent Management API service to push consent history data
The Consent Management create/update API methods could be updated to, after writing updates, either:

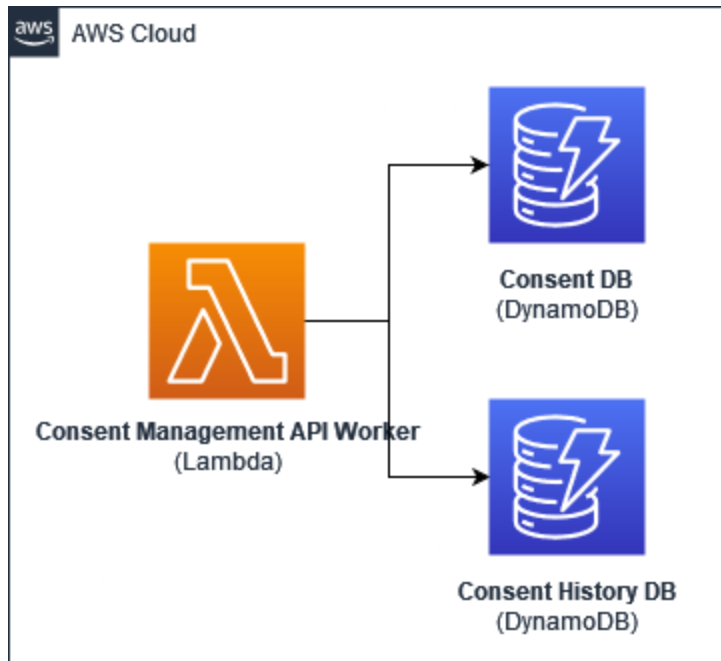
1. Directly write consent history items to DynamoDB; or
2. Invoke a Consent History Ingestor Lambda worker; or
3. Push an event to a Consent History SQS queue that triggers a Consent History Ingestor Lambda function.

Option 3.1. Consent Management API writing to Consent History DynamoDB table

The simplest approach is for the Consent Management API to both write data to the Consent DB, as well as immediately afterwards writing to the Consent History DB.

This removes the need for a stand-alone Consent History Ingestor Lambda function, and has the lowest infrastructure costs and set-up costs.

However, would rule this out as not fault-tolerant or replayable if API workers fail between writing consent data and writing consent history to DynamoDB, in which case our Consent History DB will remain incomplete and inaccurate.

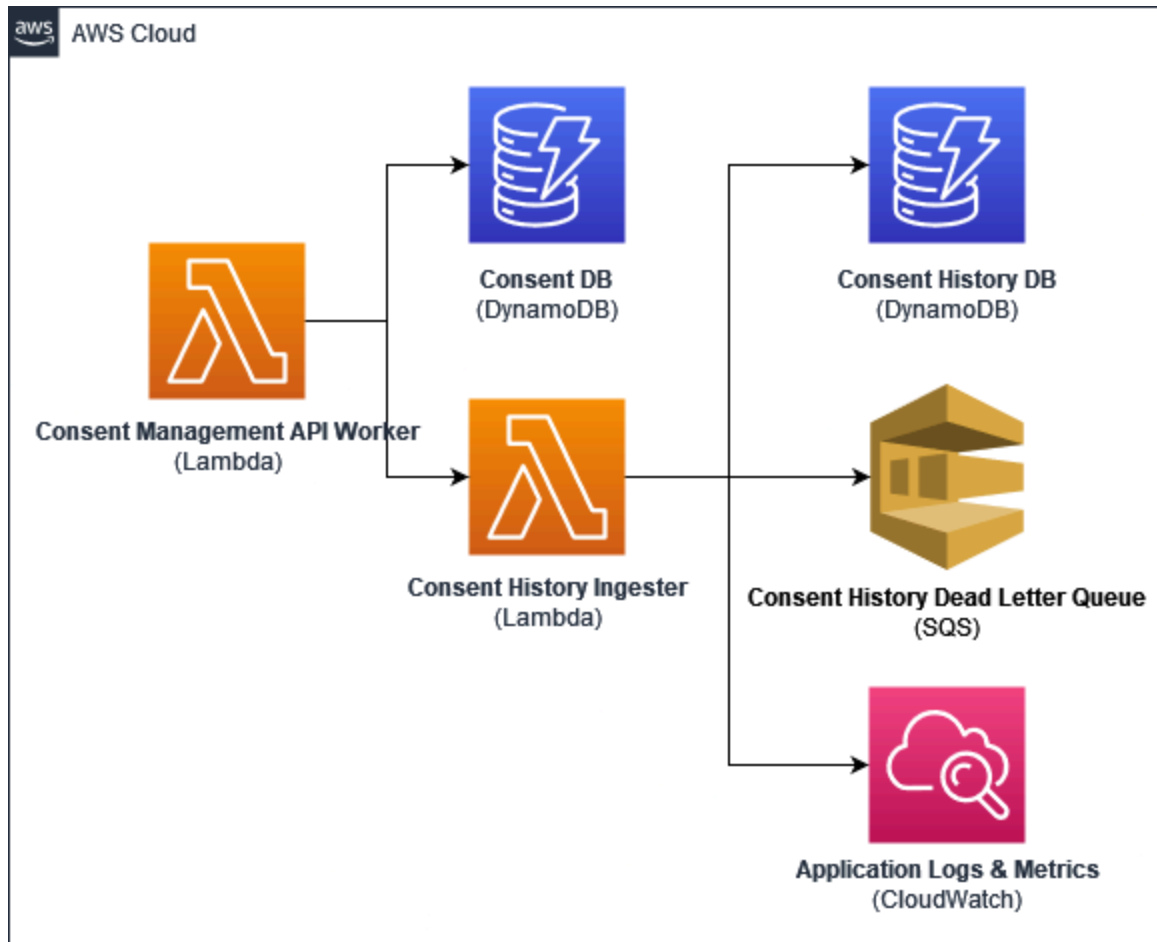


Option 3.2. Consent Management API invoking Consent History Ingestor Lambda function

If pushing events to another Lambda, also need to consider fault tolerance as well as latency to invoke the Lambda.

Would rule out synchronous calls to the Consent History Ingestor Lambda function as adding unnecessary latency and points of failure to customer-facing consent update calls.

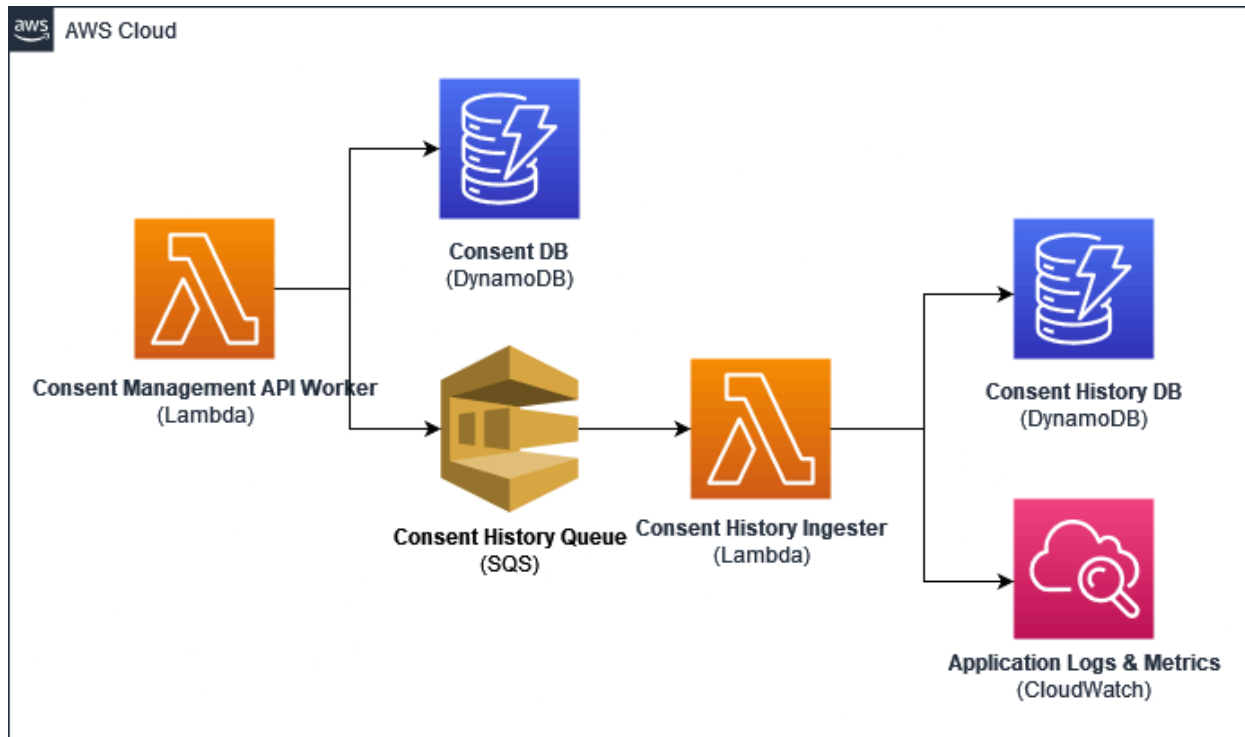
Async Lambda invocations may fail without being easily replayable, so would want to configure a SQS dead letter queue that can use to replay failed invocations.



Option 3.3. Consent Management API pushing messages to a Consent History SQS queue

The Consent Management create/update API methods could be updated to push consent update events to SQS which can subsequently be picked up by the Consent History Ingester Lambda function.

This approach is less likely to fail in a non-replayable way than Options 3.1 and 3.2, but API workers may still fail before successfully pushing the SQS messages, and this approach adds more delay to syncing consent updates to the Consent History database.



Common trade-offs

All options in this category have some fault tolerance issues if the API workers fail between writing consent data and writing or pushing Consent History events.

Advantages:

1. Reduces infrastructure, set-up effort, and associated costs of DynamoDB Streams or Kinesis Data Streams, and cost of additional Lambda runtime should be minimal for invoking an async Lambda function or pushing a message to SQS.

Disadvantages:

1. Less fault tolerant than data streams. Lose data without an easy replay mechanism if the Consent API Lambda fails between writing consent updates and pushing Consent History data.
2. Increases complexity of Consent Management API service.
3. Increases customer-facing latency of Consent Management create/update API methods.

Recommend DynamoDB Streams or Kinesis Data Streams over the Consent Management API writing Consent History data, for increased fault tolerance and lower customer-facing latency and points of failure, where DynamoDB Streams would be the lower-cost alternative.

Consent history content

The key attributes we generally want to include in a Consent History record are:

1. What changes were made

2. When those changes were made
3. Who made those changes

In our case, the “who” is implied by the `userId` of each consent, and we will want to capture what changes were made and when for each consent update.

There are multiple ways to implement a historical record, where items may:

1. Contain the updated field names and values from every write operation; or
2. Contain the diff between the old and new versions of data for every write operation; or
3. Contain a full copy of every version of data; or
4. Contain a full copy of before-and-after versions of data for every write operation.

We typically want to work backwards from a desired user experience to determine which approach makes most sense. Given we want to support arbitrary customer use cases, the 3rd and 4th options make the most sense. Option 4, storing full before-and-after versions for every write operation, is most general-purpose, while duplicating data in the Consent History database. Option 3 is simpler and easier to maintain independent of backend implementation

DynamoDB Streams support streaming any of the following options for each DynamoDB write operation that changes table item data (ref:

https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_StreamSpecification.html):

1. **KEYS_ONLY** - Only the key attributes of the modified item are written to the stream.
2. **NEW_IMAGE** - The entire item, as it appears after it was modified, is written to the stream.
3. **OLD_IMAGE** - The entire item, as it appeared before it was modified, is written to the stream.
4. **NEW_AND_OLD_IMAGES** - Both the new and the old item images of the item are written to the stream.

DynamoDB Stream events include event IDs, event names (INSERT/MODIFY/DELETE, where only INSERT and MODIFY are relevant to our use cases), event timestamps, and additional attributes depending on the chosen stream view type.

Option 3: store full copy of every version of consent data

NEW_IMAGE would be the DynamoDB stream view type we would choose to store updated data snapshots for every write operation.

Consent DynamoDB Stream events would be converted to Consent History DynamoDB items with the following data format:

```
{
```

```

    "id": "string", // Associated consent table partition key
    "serviceId": "string",
    "userId": "string",
    "consentId": "string",
    // Required consent attributes
    "consentVersion": "integer",
    "status": "string (enum values: ACTIVE, EXPIRED, REVOKED)",
    // Optional consent attributes
    "consentType": "string",
    "consentData": "object (up to 32 string key-value pairs)",
    "expiryTime": "string (ISO 8601 datetime)",
    // Event metadata
    "eventId": "string",
    "eventType": "string (enum values: INSERT, MODIFY)",
    "eventTime": "string (ISO 8601 datetime)"
}

```

Option 4: store full copy of before-and-after versions of data for every consent update

NEW_AND_OLD_IMAGES would be the DynamoDB stream view type we would choose if we wanted to store before-and-after data snapshots for every write operation.

Consent DynamoDB Stream events would be converted to Consent History DynamoDB items with the following data format:

```

{
    "id": "string", // Associated consent table partition key
    "serviceId": "string",
    "userId": "string",
    "consentId": "string",
    "eventId": "string",
    "eventType": "string (enum values: INSERT, MODIFY)",
    "eventTime": "string (ISO 8601 datetime)",
    "oldData": "object", // If applicable, contains old non-identifier
attribute values
    "newData": "object" // Contains new non-identifier attribute values
}

```

This will enable future searches across multiple dimensions such as consent id, or serviceId, or userId, or { serviceId, userId } pairs, with API clients able to decide whether to display full before-and-after data, latest data only per write operation, or a generated diff per write operation.

Recommendation

Recommend going with Option 4: store a full copy of the before-and-after snapshots of consent data for each consent update. This will allow us to support arbitrary future customer use cases without requiring new complexity in the consent history service.

Consent expiration

We want to support users setting expiration times for their consents, where our platform should automatically update those consents to have an “EXPIRED” status after they have passed their expiryTime.

We should not want to automatically delete the consents from our data store, since expired consents should still be queryable through the Consent Management API. Therefore we will not be using DynamoDB’s built-in “Time To Live” functionality ([ref](#)), which automatically deletes records that pass their TTL timestamp.

Instead, we will build a custom workflow that periodically checks for “ACTIVE” consents past their expiryTime, and updates their status to “EXPIRED”.

Recommendations:

1. Start with a CloudWatch Event based cron schedule that triggers an hourly Lambda job for the lowest-effort, lowest-cost v1 solution.
2. Set up a GSI that uses expiry hour as its partition key and expiry time + id as its sort key.
 - a. Update consent write APIs to set these values for active consents with expiry times, and clear the values if the consent status != “ACTIVE” or if expiry time is null.
 - b. Implement Consent Expiry Processor logic to query the GSI for every hour from several days ago to one hour ago and update all matching consents to expired.
 - c. Implement Consent Expiry Processor logic to query the GSI for the current hour, and update each matching consent to expired until reach a consent that has a future expiry time and then short circuit.
3. Emit business CloudWatch metrics from the auto-expiration job to monitor how many consents expire over time, and monitor when we start falling behind expiryTime values by a time period approaching our SLA.
4. If we start to fall behind and need to increase frequency or add concurrent time buckets such that auto-expiration Lambda runtimes are over 12 instance hours per day, migrate to a continuous polling auto-expiration service which will at that point be less expensive.

Compute and scheduling infrastructure options

Recommendation: CloudWatch Event + Lambda Function

- Simplest and lowest-cost to set up for v1 launch.

- Sufficient to start with an hourly schedule for invoking the Lambda, can monitor metrics to evaluate when we reach a scale where it makes sense to migrate to a continuous polling service hosted in ECS/EC2.

Option 1: Periodic CloudWatch Event + AWS Lambda Function

If the scale of expiring data is low and all expiring consents can be processed within a 15 minute job runtime, a simple approach is to:

1. Create a new *AutoConsentExpiryProcessor* Lambda function that:
 - a. Queries an *ActiveConsentsWithExpiryTime* GSI for all active consents with expiryTime values less than the current time
 - b. For all matching records, submit UpdateItem requests to the *ServiceUserConsent* DynamoDB table where partition key id = the GSI record's GSI partition key value, setting status = "EXPIRED" and setting the GSI partition key attribute value to null to remove it from the GSI.
2. Set up a periodic CloudWatch Event that uses a cron expression to trigger on a fixed schedule, and have it invoke the *AutoConsentExpiryProcessor* Lambda function.

Pricing estimates:

- While our platform's total Lambda usage is under the 1M requests/month and 3.2M seconds compute time/month free tier, cost is \$0.
- Once we exceed the free tier from other platform workflows, Lambda charges \$0.20/M requests and \$0.00001667/GB-second.
 - If we run the Lambda function hourly for the maximum 15 minutes, and allocate 1 GiB memory, $1 \text{ GiB memory} * 15 \text{ minutes runtime/hour} * 24 \text{ hours/day} * 365/12 \text{ days/month} * 60 \text{ seconds/minute} * \$0.00001667/\text{GB-second} = 15 * 24 * 365 / 12 * 60 * 0.00001667 = \$10.95/\text{month}$ for compute time. Request charges will be $1 * 24 * 365 / 12 * 0.20 / 1000000 = \0 , for a total of \$10.95/month.
 - If we run the Lambda function every 1 hour for an average 10 minute runtime, request charges are still effectively \$0 and compute time charges are $10 * 24 * 365 / 12 * 60 * 0.00001667 = \$7.30/\text{month}$.
 - If we run the Lambda function every 3 hours for an average 10 minute runtime, request charges are still effectively \$0 and compute time charges are $10 / 3 * 24 * 365 / 12 * 60 * 0.00001667 = \$2.43/\text{month}$
- Expect the cost to be free for the foreseeable future, remaining under \$11/month even if we scale to receive > 1M requests/month across our platform's Lambda functions and run the auto-expiration job hourly for the max 15 minute runtime/hour.

Pros:

- Easy to implement, can migrate to longer lived compute types in the future if needed.
- Effectively free for the immediate future, and even at scale continues to be very low-cost if total runtime is a small fraction of the day.

Cons:

- Limited to Lambda's 15 minute max runtime, which may not be sufficient if have too many expiring records to process within that time. Do not expect this to be an issue on initial launch, only if/when we onboard customers with very large scale data.
- May be higher cost than ECS if need to run very frequently to auto-expire records within a few minutes of their expiry time.
- Will have throughput limits if only running a single Lambda function instance at a given time to avoid collisions on the same records.
 - Could potentially scale solution to run multiple instances concurrently if they have disjoint time periods they scan for, eg. one Lambda instance running hourly to auto-expire any records that expired within the last hour, and a concurrent Lambda instance that auto-expires any records that expired > 1 hour ago, potentially with some buffer between buckets to eliminate any potential overlap, eg. if have 15 minute gap between buckets, should be sufficient given Lambda's max 15 minute runtime.
 - If needed to add additional buckets, could have 1 hourly job to auto-expire records that expired within the last hour, 1 hourly job to auto-expire records that expired 61-120 minutes ago, 1 hourly job to auto-expire records that expired > 120 minutes ago, etc.
 - If we end up effectively running Lambda for more than half of every day, it will be more cost-effective to have a continuous EC2 instance polling and auto-expiring consents.

Ref:

<https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/scheduled-events-invoking-lambda-example.html>

Option 2: Periodic CloudWatch Event + ECS task

If the scale of expiring consents is too large to process within Lambda's 15 minutes max runtime, but we don't have a tight SLA and can auto-expire within 3-6+ hours of the expiryTime, we could follow the same approach as Option 1 but with an ECS task, which does not have any max runtime limitations.

Pricing estimates:

- If run 1 ECS/Fargate task with 1 vCPU, 2 GiB RAM, in us-west-2, with ARM processor (lower cost and more efficient), and it runs on average 30 minutes/day, this costs \$0.60/month.
- If run the same task every 3 hours with an average of 10 minutes/run, 10 minutes/run * 24/3 runs/day = 80 minutes/day, this costs \$1.61/month.
- If run the same task every 1 hour with an average of 10 minutes/run, 10 minutes/run * 24 runs/day = 240 minutes/day, this costs \$4.81/month.

Pros:

- Relatively straightforward to implement, potentially a week of additional set-up effort over Lambda functions if using ECS Fargate.

- Add a few days on top of this if using EC2 on ECS, but Fargate is preferred over EC2 for lower set-up and maintenance effort, and is comparable in cost if the service isn't running for most of the day.
- Low-cost if infrequently run and if total runtime is a small fraction of the day.

Cons:

- Higher effort to set up and maintain than Lambda.
- May be higher cost than EC2 if need to run very frequently to auto-expire records within a few minutes of their expiry time.

Option 3: Continually running ECS/EC2 service that polls for expiring consents

If we want to support SLAs within a few minutes for auto-expiring consents, we could set up a permanently running service that polls for expiring consents and updates their statuses.

This works well while the scale of expiring records is small enough for a single host to clear within our chosen SLA, while a single host will have throughput limitations even with multi-threading, and if we scale to multiple hosts they may have collisions attempting to concurrently process the same records.

Within a single, long-running host, we can batch expiring consents into multiple threads to submit concurrent DynamoDB UpdateItem API requests until they have all been updated.

Since this will be a very lightweight process that only needs enough memory to process a single page of DynamoDB results at a time, and each GSI record will only need to have a couple of string attributes, we can use a very small EC2 instance type such as t4g.micro which has 2 vCPUs and 1 GiB RAM.

Price estimates:

- If continually running 1 t4g.micro EC2 instance in us-west-2, on-demand costs will be $\$0.0084/\text{instance hour} * 24 \text{ hours/day} * 365 \text{ days/year} = \$73.58/\text{year}$.
- If paying upfront for 1 year's usage via AWS' Savings Plan (not reserved instance), the cost will be $\$0.0057/\text{instance hour} * 24 * 365 \text{ hours/year} = \$49.93/\text{year}$.
- If paying upfront for a 1-year reserved instance, the cost will be \$43/year, while we'll be locked into this instance type.

Pros:

- Relatively straightforward to implement.
- Allows tighter SLA than scheduled jobs for auto-expiring records, so long as a single host has sufficient throughput for expected scale of expiring consents.

Cons:

- Likely 1-2 weeks more effort to set up than a periodic Lambda function, higher maintenance than Lambda.
- Higher cost than periodic Lambda or ECS/Fargate tasks.

- Scaling limits / bottleneck imposed by either having a single host, or by potentially colliding and processing the same records concurrently if scaling to multiple hosts.
 - Should be fine with a single low-cost instance while < 100 consents expire per second.

Database integration options

Recommendation: (Option 4) GSI with expiry hour as partition key and expiry time + id as sort key

- Provides best query efficiency and scalability out of the considered options.
- Enables DynamoDB GSI query results to be automatically sorted in ascending order of expiry time, allowing us to efficiently process consents expiring in the current hour and short-circuit once we reach a consent expiring in the future.
 - Option 1 requires scanning all consents with active expiry times, which is a blocking issue.
 - Options 2-3 have issues with needing to scan all consents expiring in the current bucket.

Option 1: GSI with autoExpireId partition key and expiryTime sort key

For a simple way to track active consents with expiry times, we could:

1. Update the ServiceUserConsent DynamoDB table to have an *ActiveConsentsWithExpiryTime* GSI with a new sparse GSI partition key that is programmatically set for active consents with non-null expiryTime values, and the sort key set to expiryTime.
 - a. This GSI can be a projection that only includes the GSI partition key, expiryTime, and consentVersion (needed for update consent API requests), since other attributes are not needed, and this minimization will reduce query latency and storage + query costs.
2. Update the Consent Management API to set the GSI partition key attribute to the consent's partition key whenever creating or updating a consent to have status = "ACTIVE" and expiryTime to a non-null value.
3. Have the Consent Expiry Processor Lambda Function make paginated DynamoDB Scan queries against the *ActiveConsentsWithExpiryTime* GSI to process all consents where expiryTime is in the past.

Pros:

- Relatively small scope of service changes needed across APIs/DBs to support use cases.
- Simple mental model for identifying and updating consents that need to be expired.

Cons:

- **DynamoDB scans do not support ordering by sort key**, so a GSI scan would require scanning the entire index to identify all consents past their expiry time.
 - This will become very inefficient if expiry times are frequently used.

- Suggest considering alternative GSI/table structures that are more scalable for supporting our use cases.

Option 2: GSI with expiry date as time-bucketed partition key, consent table partition key as sort key

If we wanted to be able to leverage built-in DynamoDB functionality to efficiently look-up consents with expiry times in the past, we could set up a DynamoDB GSI with time-bucketed expiry time partition keys.

We would then use the consent's table partition key (`id`) as the sort key, since we need to enforce that every GSI item has a unique GSI partition key + sort key combination.

This would allow us to efficiently query for all consents in a given time bucket that we know in advance is in the past, while we would need some way to predict which time buckets have GSI items.

A simplified approach would be to use the expiry date (eg. "2025-04-13") as the time bucket partition key, and have the Consent Expiry Processor query for every date in the last week, with the expectation that we will have alarms set up to notify ourselves of any issues and fix them within a week, or temporarily expand the searched date range if needed after an incident that takes longer than a week to fix.

Then the Consent Expiry Processor workflow will be:

1. For each date in the last week up to yesterday:
 - a. Query the *ActiveConsentsWithExpiryTime* GSI for all consents with an expiryTime on the given date.
 - b. For all matching consents, update their status to "EXPIRED", increment their version, and clear their *ActiveConsentsWithExpiryTime* GSI partition key value.

Scaling limits:

- As per <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/TroubleshootingThrottlingOnDemand.html>, "Each partition of a table or GSI can serve up to 3,000 read request units or 1,000 write request units per second. If traffic to a partition exceeds this limit, the partition might be throttled."
- As per <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/read-write-operations.html>, "A strongly consistent read request of an item up to 4 KB requires one read unit. An eventually consistent read request of an item up to 4 KB requires one-half read unit."
- Assuming we make eventually consistent reads for efficiency and conditional write expressions when updating consents to expired for safety, our scaling limit will be querying up to **6,000 consent records per expiry time bucket per second**.

- If a single paginated DynamoDB query against an expiry time bucket (eg. the bucket for consents expiring yesterday) returns 1000 consent records, this query will consume 500 read units.
- **When tens of thousands of consents expire on the same day**, bucketing expiry times at the date granularity may hit throttling limits, and will need to space out requests, or retry after a wait time when catch throttling errors.

Pricing:

- DynamoDB charges \$0.125 per million read request units.

Pros:

- Relatively small scope of service changes needed across APIs/DBs to support use cases.
- Simple mental model for identifying and updating consents that need to be expired.
- Efficient query patterns to identify all consents that need to be expired.

Cons:

- Need to query for every time bucket that could potentially have active consents with expiry times.
 - Not a major issue if time-bucket by expiry date, and just have 7 query partition keys if querying for 7 days up to yesterday.
- Will have DynamoDB hot key / throttling issues if need to scale to query > 6k records/second within the same expiry time bucket.
 - May not be an issue while relatively low scale, where can self-throttle via the Consent Expiry Processor having a single worker that processes one page of results at a time, and finish processing all expiring records within 15 minutes.
 - However, if data scales to have tens of thousands of consents expiring on the same day, may need to make time buckets more granular, eg. hourly time buckets.
- Limited granularity of partition keys means that we will either only be able to auto-expire consents a day after they've expired, or that we'll need to manually check the expiry time of every consent that expires today to only expire those that are in the past.
 - This isn't a huge issue, just need branching logic for the current date's bucket, to sort each page of results by expiryTime and skip to the next page whenever we hit a consent that expires at a future time today.

Option 3: GSI with expiry hour as time-bucketed partition key, consent table partition key as sort key

As with Option 2, this option would set up a DynamoDB GSI with time-bucketed expiry time partition keys, and the consent `id` as the sort key, while in this option we would use the **expiry hour** as the partition key.

This will provide more granular DynamoDB partitions that are less likely to hit scaling issues, as we will be able to query up to 6k expiring consents per second that have the same expiry hour.

Then for each hour in the last week up to 1 hour ago, the :

1. Query the *ActiveConsentsWithExpiryTime* GSI for all consents with an expiryTime in the given hour.
2. For all matching consents, update their status to "EXPIRED", increment their version, and clear their *ActiveConsentsWithExpiryTime* GSI partition key value.

Scaling limits:

- Can query up to 6k consents/expiryHour/second without DynamoDB throttling.

Pros:

- Relatively small scope of service changes needed across APIs/DBs to support use cases.
- Simple mental model for identifying and updating consents that need to be expired.
- Efficient query patterns to identify all consents that need to be expired.

Cons:

- Need to query for every time bucket that could potentially have active consents with expiry times.
 - Not a major issue given DynamoDB queries are fast and low-cost for the maximum number of queries without results we're considering.
 - Assuming querying across the last 7 days, with 24 expiry time buckets per day, this would be $7 \times 24 = 168$ partition keys for the job to query DynamoDB for per job run, with additional paginated queries for the buckets that have multiple pages worth of expiring consents.
 - DynamoDB queries against empty partitions should be very fast, assume $10\text{ms/query} \times \text{minimum } 168 \text{ queries/job run} \times 1\text{s}/1000\text{ms} = \text{up to } 1.68 \text{ seconds/job run}$ for querying empty partitions.
 - Assuming our in-service code adds $\sim 30\text{ms}$ to submit each DynamoDB query and process empty results, this is still $40\text{ms/query} \times 168 \text{ queries/job run} \times 1\text{s}/1000\text{ms} = 6.72 \text{ seconds/job run}$ for querying empty partitions, not an issue.
 - If run the job once per hour, $24 \text{ job runs/day} \times \text{minimum of } 168 \text{ DynamoDB queries per job run to query each time bucket} \times 365/12 \text{ days/month} = \text{up to } 122,640 \text{ unnecessary DynamoDB queries/month} \times 1 \text{ read request unit}/2 \text{ queries} \times \$0.125/10^6 \text{ read request units} \sim \$0.0077/\text{month} \rightarrow \text{rounds to } \$0.01/\text{month}, \text{ effectively free.}$
- Will have DynamoDB hot key / throttling issues if need to scale to query $> 6\text{k}$ records/second that have the same expiry hour.
 - Don't expect this to be an issue for a while, can always add minor wait time between paginated queries for the same expiry hour.
- Will either only be able to expire consents 1-2 hours after they've expired, or need special code handling for the current expiry hour's partition key to manually check all

consents expiring in the current hour to only expire those where expiryTime is in the past.

- Same logic as with Option 2, but also lower priority in this option given the time buckets are at the hourly granularity and it's less problematic to have a 2-hour SLA than to have a 1-day SLA for expiring consents.

Option 4: GSI with expiry hour as partition key, expiry time + id as sort key

In order to support efficiently sorting query results by expiry time, we could set up a DynamoDB GSI called *ActiveConsentsWithExpiryTime* with the expiry hour as the GSI partition key, and the expiryTime + "|" + the consent's id value as the GSI sort key.

This would result in every query's results automatically being sorted by expiry time, and allow us to efficiently query the current hour's partition and short-circuit whenever we hit a consent with an expiry time in the future, without needing to query and process every page of results in the current hour's partition.

As per

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.KeyConditionExpressions.html>: *"Query results are always sorted by the sort key value. If the data type of the sort key is **Number**, the results are returned in numeric order. Otherwise, the results are returned in order of UTF-8 bytes. By default, the sort order is ascending. To reverse the order, set the **ScanIndexForward** parameter to **false**."*

Pros:

- Relatively small scope of service changes needed across APIs/DBs to support use cases.
- Simple mental model for identifying and updating consents that need to be expired.
- Efficient query patterns to identify all consents that need to be expired, including for consents expiring in the current hour's time bucket.

Cons:

- Slightly more API work needed than with prior options to set two GSI-specific fields, expiry hour and expiryTime + "|" id.
- Need to query for every time bucket that could potentially have active consents with expiry times.
 - Not a major issue given DynamoDB queries are fast and low-cost for the maximum number of queries without results we're considering.
- Will have DynamoDB hot key / throttling issues if need to scale to query > 6k records/second that have the same expiry hour.
 - Don't expect this to be an issue for a while, can always add minor wait time between paginated queries for the same expiry hour.

API client config management

▣ Design for dynamic API client config in AWS

Programming language

Main two programming languages under consideration are Java and TypeScript.

- Java and TypeScript have first-class support for backend AWS applications, both have mature, well-documented AWS ecosystems
- Ruling out Rust and Go as having higher learning curves and more difficult to find skilled developers for, while would be interesting for performance-critical applications
- Ruling out Python for lower performance than all other languages considered, low memory efficiency, lack of type safety
- For defining CDK infrastructure, TypeScript is the best supported CDK language

Either Java or TypeScript are excellent choices for service code.

- Team familiarity and best-supported programming languages within a company/team will be key factors for productivity.
- If need to minimize latencies, may be useful to benchmark specific services' use cases in both languages to get a more relevant performance comparison, while either will work for our use cases.

Recommendation: TypeScript for CDK, Java for service code

- Both Java and TypeScript are excellent choices, team familiarity and best-supported languages within a company/team will be key factors for productivity
- Prefer TypeScript for CDK packages defining AWS infrastructure since TypeScript is the best supported language for CDK
- For service code, slightly prefer Java for ease of onboarding team members
- Optimized Java functions can be more performant than TypeScript/NodeJS, and plan to have "canary" tests periodically pinging the API service to monitor availability and keep Lambda function warm which should minimize Java's cold start issues

	Rust	Go	Java	TypeScript	Python
Performance	High performance Much lower floor on latency than Java, TypeScript, or Python	High performance Not quite as fast as Rust, much more performant than Java, TypeScript, or Python	Much less efficient than Rust or Go. Optimized Java apps can be faster than Typescript/Python for warmed up Lambdas, while cold start time is much slower Can mitigate Lambda cold starts via "canary"/ping calls every few minutes	Much less efficient than Rust or Go. Comparison with Java/Python depends on use case, often lower start-up time than Java for rarely called functions	Much less efficient than Rust or Go Comparison with Java and TypeScript depends on use case, benchmarks are inconsistent while https://blog.scanner.dev/serverless-speed-rust-vs-go-java-python-j-n-aws-lambda-functions/ shows Python as significantly slower at processing 1 GB of data from S3

					Java can be more efficient for long-lived applications or frequently called Lambda functions, while Python can have very low start-up times for rarely called, short-lived functions Can be more efficient when only using optimized C++ based libraries
Learning curve	High Many new concepts, lower-level than most modern languages and more difficult to ramp up on, not as mature or well-documented as others Difficult to find developers familiar with Rust and takes more time than other languages to learn	Moderate Lower-level than Java/TypeScript/Python, less mature ecosystem Somewhat difficult to find developers familiar with Go, new developers less productive than in Java, TypeScript, or Python	Low Very mature, well-documented ecosystem, easy to ramp up new developers	Low Easy to pick up if have worked with other statically typed languages	Low Very easy to ramp up on, widely used
Strongly typed	Yes	Mostly, has duck-typing	Yes	Optional, can specify types, plain JavaScript libraries/calls are not typed	No
Ease of cross-platform development	Reasonable for non-GUI applications	High	High	High	High
Memory efficiency	High	High	Moderate Performance highly dependent on available memory, see significant latency improvements when allocating at least 1 GB memory	Moderate	Not very memory efficient
Suitability for backend AWS applications	Moderate, with caveat of being fairly new to AWS and not as well supported/documented as Java, TypeScript, or Python, limited projects to reference Best suited to applications with very low latency requirements	Moderate, with caveat of being fairly new to AWS and not as well supported/documented as Java, TypeScript, or Python Best suited to applications with very low latency requirements	High, very well-supported and documented, most mature AWS SDK	High, very well-supported and documented	Moderate, less mature and less optimized AWS SDK than for Java and TypeScript, not as many AWS services built using Python
Ideal application	Performance-critical applications, system-level programming	Networking applications, concurrent programming	Enterprise applications, Android applications	Front-end applications	Data analytics, machine learning, scientific computing

Integration tests

Summary of considered approaches

	CDK integ-tests construct	AWS IATK	Unit test framework (eg. JUnit) + AWS SDK	Unit test framework (eg. JUnit) + HTTP client	AWS CLI
Summary	CDK constructs simplifying AWS integ tests, API calls, validations, retry logic	Library simplifying integ tests for event-based architectures	Implement JUnit tests without mocking, use AWS SDK to query and validate services	Implement JUnit tests without mocking, use HTTP client to query and validate externalized APIs	Implement CLI based tests that invoke the API Gateway TestInvokeMethod API to query the APIs with built-in auth
Initial set-up effort	Low	Medium-high (need to set up supporting infra/integrations)	Medium	Medium-high (need to set up custom SigV4 integration)	Low
Maintenance effort	Unknown, may need to migrate off if deprecated	Unknown, may need to migrate off if deprecated	Low	Low	Medium-low (less easy to extend / reuse code compared to JUnit)
Supports integ tests against API services	Yes	Not directly <i>Need to set up intermediate Lambda function that encapsulates logic for invoking the actual APIs, then use this library to invoke the Lambda.</i>	Yes	Yes	Yes
Supports integ tests against backend-only workflows w/o externalized APIs	Yes	No	Yes	No	Yes
Stability	Experimental / Alpha	Public Preview	Stable	Stable	Stable
Language support	TypeScript, Java, Python, Go, NET	Python only	Wide set of languages including any we'd consider	All programming languages	Bash

Recommendation: In the long term, use JUnit + AWS SDK, in the short term can start with GitHub Workflows + AWS CLI

- Both:
 - Support all current and future expected integration test use cases
 - Avoid depending on experimental/preview frameworks that may have limited support and be deprecated or have breaking changes in subsequent releases
- JUnit + AWS SDK:
 - Can reuse code from unit tests, just have separate test suites that make actual service calls without mocks
 - Once set up, easier to extend and run locally
- GitHub Workflows + AWS CLI:
 - Easy to integrate with GitHub-based CD/CI pipeline
 - Much easier to initially set up than other options

- Spent several nights trying to get either the API Gateway generated Java client working and migrated to AWS SDK for Java 2.x (doesn't build out-of-the-box, on very old deprecated libraries that have breaking changes in newer versions), or set up custom SigV4 signing with the HTTP requests in the OpenAPI Generator based Java client code (having consistent auth errors that are difficult to root-cause), without success.
- Was able to get AWS CLI based tests working within a night, much faster to validate via local CLI commands and then set up in GitHub Workflows.
- AWS CLI has built-in SigV4 auth that *just works*, compared to the stripped down HttpRequest objects available with OpenAPI Generator based Java API client where need to implement custom SigV4 auth integrations.
- Not a proper test framework but good enough for now, can create a backlog item to migrate to JUnit + AWS SDK.

AWS CDK Integration Testing Framework

AWS Cloud Development Kit (CDK) provides constructs for defining and running integration tests. The `integ-tests` CDK construct allows you to define test cases, and the `integ-runner` automates resource provisioning and cleanup.

Pros:

- Seamless integration with AWS infrastructure, clean abstraction layer defined in CDK
- Simplifies making API calls to AWS components and AWS-hosted API services, validating responses/results, implementing retry logic
- Support for multiple programming languages (TypeScript, Java, Python, Go, .NET)

Cons:

- `integ-tests` CDK constructs are marked as experimental, may be dropped or have breaking changes in future releases - *Prefer not to rely on experimental constructs.*

References

- <https://docs.aws.amazon.com/cdk/api/v2/docs/integ-tests-alpha-readme.html>

AWS Integrated Application Test Kit (IATK)

A library designed for writing automated tests for serverless applications. It simplifies locating resources and creating test harnesses for event-driven architectures.

Pros:

- Simplifies testing event-driven architectures

Cons:

- Currently marked as Public Preview, may be dropped or having breaking changes in future releases

- Limited functionality, does not directly support our main use case of integration testing API services
 - Have some workarounds such as if we create a Lambda function that encapsulates calling the API, then can trigger a test event to that Lambda via AWS IATK
- Limited language support, currently only supports Python
 - Our services currently use TypeScript for CDK and Java for everything else, not ideal to add Python just for integration tests

References

- Introduction blog post: <https://aws.amazon.com/blogs/compute/aws-integrated-application-test-kit/>
- GitHub repository: <https://github.com/aws-labs/aws-iatk>

Custom integration tests using JUnit + AWS SDK

Similar to how our unit tests are implemented, we could write integration tests in JUnit using the AWS SDK to query our AWS-hosted services and validate responses. These tests should not have any mocks, so we call our actual services using permissions vended via our assumed IAM role, and account/stage-specific identifiers/endpoints passed via environment variables.

Pros:

- Highly flexible, can test any scenarios and validations supported by AWS SDK, not limited to external APIs, expect to support all current and future use cases
- Can reuse code/libraries/frameworks from service unit tests

Cons:

- Higher implementation and maintenance effort compared to prebuilt AWS test frameworks

Custom integration tests using JUnit and an HTTP client

Without depending on the AWS SDK, write integration tests in JUnit that invoke our API endpoints using HTTP requests only, and validate responses. This imitates customer interactions for the hypothetical use case where clients may have zero awareness/integrations with AWS, and want to call our services via REST APIs without depending on architecture / cloud platform choices.

Pros:

- Closest to end customer experience, more accurately validates system behaviour from customer perspective

Cons:

- Limits testability to what is supported by externalized APIs (eg. won't be able to test the history ingestor workflow until have published a consent history API)

- Requires more custom code than JUnit + AWS SDK, where could reuse existing unit test code

AWS CLI

The AWS CLI provides programmatic access to a wide variety of AWS service queries, and we can easily leverage it in GitHub Workflows to dynamically look up resource IDs via querying CloudFormation exports, then make API Gateway CLI queries to invoke each API endpoint and validate responses.

Pros:

- Highly flexible, can test any scenarios and validations supported by AWS CLI, not limited to external APIs, expect to support all current and future use cases
- Easier to get working in GitHub Workflows than setting up API Java clients.

Cons:

- Bash based integ tests are less easy to work with and extend/maintain over time compared to JUnit tests which can more easily reuse test code.
- API Gateway TestInvokeMethod based integ tests aren't exactly 1:1 to how customers will call our APIs.

Appendix 3: Pricing estimates

Notes:

- All pricing is per month, and based on the us-east-2 (Ohio) region.
- These are rough estimates based on early scale/usage projections prior to implementation. Actual costs may differ, and would tune infrastructure and estimates after setting up initial implementation and running load tests that mirror scaled down expected usage.
- Pricing estimates only consider the prod environment to give an expectation of the order of magnitude of expected spending. Would plan to have non-prod stages with much lower scale of data + activity for validating service changes.
- Purpose of this exercise is to give an early idea of which design choices are likely to be an order of magnitude more or less expensive at different scales we expect to support, to consider in design decisions.

# Customers	Total monthly spending	DynamoDB	API Gateway	API Workers	Consent History Ingestion	Web App Firewall	CloudWatch
10k	\$37.61	\$1.29	\$6.30	\$0.13	\$0	\$15.37	\$14.52
100k	\$68.81	\$12.88	\$20.65	\$2.10	\$0	\$18.67	\$14.52
1M	\$399.86	\$128.75	\$181.30	\$22.75	\$0.84	\$51.70	\$14.52
10M	\$3,731.88	\$1,323.12	\$1,664.60	\$336	\$11.64	\$382	\$14.52

Assumptions for above summary of estimated monthly spending:

- Choose DynamoDB for storing consent records and consent version history
- Choose REST API Gateway without caching for API interfaces
- Choose Lambda for API compute up to 1M end customers, then migrate to EC2 on ECS before get to 10M end customers
- Choose DynamoDB Streams + Lambda + SQS dead-letter-queue for consent history ingestion

Database service options

	Variation	Total estimated monthly costs	Data storage	Read transactions	Write transactions	Instance hours	Point-in-time backups	Data transfers out	Other charges
DynamoDB with 100k customers	Standard	\$6.50	\$0.00	\$1.25	\$0.25	\$0.00	\$5.00	\$0.00	\$0.00
	Global	\$12.88	\$6.25	\$1.25	\$0.38	\$0.00	\$5.00	\$0.00	\$0.00
DynamoDB with 1M customers	Standard	\$121.25	\$56.25	\$12.50	\$2.50	\$0.00	\$50.00	\$0.00	\$0.00
	Global	\$128.75	\$62.50	\$12.50	\$3.75	\$0.00	\$50.00	\$0.00	\$0.00
DynamoDB with 10M customers	Standard	\$1,302.66	\$618.75	\$125.00	\$25.00	\$0.00	\$500.00	\$33.91	\$0.00
	Global	\$1,323.12	\$625.00	\$125.00	\$37.50	\$0.00	\$500.00	\$35.62	\$0.00
Aurora with 100k customers 1 master + 2 replicas	Standard	\$140.49	\$7.50	\$1.00	\$0.00	\$131.40	\$0.59	\$0.00	\$0.00
	Standard Global	\$140.53	\$7.50	\$1.00	\$0.04	\$131.40	\$0.59	\$0.00	\$0.00
	I/O Optimized	\$192.68	\$16.89	\$0.00	\$0.00	\$175.20	\$0.59	\$0.00	\$0.00
	I/O Optimized Global	\$192.70	\$16.89	\$0.00	\$0.02	\$175.20	\$0.59	\$0.00	\$0.00
Aurora with 1M 1 master + 2 replicas	Standard	\$1,146.54	\$75.00	\$10.00	\$0.20	\$1,055.49	\$5.85	\$0.00	\$0.00
	Standard Global	\$1,146.74	\$75.00	\$10.00	\$0.40	\$1,055.49	\$5.85	\$0.00	\$0.00
	I/O Optimized	\$1,576.20	\$168.75	\$0.00	\$0.00	\$1,401.60	\$5.85	\$0.00	\$0.00
	I/O Optimized Global	\$1,576.40	\$168.75	\$0.00	\$0.20	\$1,401.60	\$5.85	\$0.00	\$0.00
Aurora with 10M 1 master + 2 replicas	Standard	\$11,499.69	\$750.00	\$100.00	\$2.00	\$10,554.90	\$58.50	\$34.29	\$0.00
	Standard Global	\$11,503.40	\$750.00	\$100.00	\$4.00	\$10,554.90	\$58.50	\$36.00	\$0.00
	I/O Optimized	\$15,798.00	\$1,687.50	\$0.00	\$0.00	\$14,016.00	\$58.50	\$36.00	\$0.00

	I/O Optimized Global	\$15,800.00	\$1,687.50	\$0.00	\$2.00	\$14,016.00	\$58.50	\$36.00	\$0.00
PostgreSQL RDS with 100k "" 1 master + 2 replicas	db.m6g.large General purpose, Graviton, 2 vCPU, 8 GiB	\$716.17	\$17.25	\$0.00	\$0.00	\$696.42	\$2.50	\$0.00	\$170.82
PostgreSQL RDS with 1M "" 1 master + 2 replicas RDS proxy	db.m6g.4xlarge General-purpose, 16 vCPUs 64 GiB memory	\$5,768.86	\$172.50	\$0.00	\$0.00	\$5,571.36	\$25.00	\$0.00	\$630.72
PostgreSQL RDS with 10M "" 1 master + 2 replicas	db.r6g.16xlarge Memory-optimized 64 vCPUs, 512 GiB memory	\$33,518.15	\$1,725.00	\$0.00	\$0.00	\$31,507.53	\$250.00	\$35.62	\$2,207.52

Source pricing spreadsheet, based on estimates in below sections:

https://docs.google.com/spreadsheets/d/1No6YfG-97Q4AxWYA85V03vv2I7xcpFdY_wqY-N-fE-4/edit?usp=sharing

Estimated DynamoDB costs/month

- Data storage: First 25 GB free (if using standard tables), then \$0.25/GB/month
- Read transactions: \$0.25/million RCUs * 1 RCU/strongly consistent read = \$0.25/millions reads given we will prefer strongly consistent reads for the consent management platform
- Write transactions:
 - Different pricing depending on whether use standard tables or DynamoDB Global Tables
 - Global Tables replicate data across multiple global regions to improve availability from 99.99% to 99.999% and reduce latencies for global customers by providing data closer to any given customer
 - Global table pricing: \$1.875/million replicated WCUs * 2 WCUs/transaction write (assuming consent items <= 1KB) = \$3.75/million writes
 - Standard table pricing: \$1.25/million WCUs * 2 WCUs/transaction write ("") = \$2.50/million writes
- Point-in-time backups: \$0.20/GB/month
- Data transfer in: \$0
- Data transfer out: First 100 GB free, then \$0.09/GB for first TB, discounts for further data out
 - For Global Tables, transfer out from master to 2 replicas in different regions for each write

- For the 1M customer scenario, $1\text{M writes/month} * 2 \text{ replica data transfers/write} * 1\text{KB/transfer} * 1/1024 \text{ MB/KB} * 1/1024 \text{ GB/MB} \approx 1.9 \text{ GB/month}$
 - For Standard and Global, assume all read queries from companies and end users trigger data transfers out, $50\text{M reads/month} * 1 \text{ data transfer/read} * 1\text{KB/transfer} * 1/1024 \text{ MB/KB} * 1/1024 \text{ GB/KB} = 47.68 \text{ GB}$
 - Assuming the majority of data out to customers comes from DB -> API -> customers, this remains within the free tier.
 - For the 10M customer scenario, $47.68 + 1.9 \text{ GB} * 10 = 495.8 \text{ GB} - 100 \text{ GB free} = 395.8 \text{ GB charged} * \$0.09/\text{GB} = \$35.62$
- Ref: <https://aws.amazon.com/dynamodb/pricing/on-demand/>

Estimated Aurora Serverless v2 costs/month

- Instance hours:
 - Standard: \$0.12/ACU hour
 - For 1M customer scenario:
 - If constantly maintain 5 ACUs, $5 \text{ ACUs} * \$0.12/\text{hour} * 24 \text{ hours/day} * 365/12 \text{ days/month} = \$438/\text{month}$
 - If able to scale down by up to 50% during non-peak usage, and run on average 4 ACUs monthly cost is $4 * \$0.12 * 24 * 365/12 = \$350.40/\text{month}$ per DB instance
 - Given 1 master + 2 read replicas, will spend $\$350.40/\text{instance} * 3 \text{ instances} = \$1,055.49/\text{month}$
 - For 100k customer scenario, must maintain at least 0.5 ACUs/instance * 3 instances * $\$0.12 * 24 * 365/12 = \131.40 minimum spend
 - I/O-optimized: \$0.16/ACU hour
 - For 1M customer scenario:
 - If constantly maintain 5 ACUs, $5 \text{ ACUs} * \$0.16/\text{hour} * 24 \text{ hours/day} * 365/12 \text{ days/month} = \$584/\text{month}$
 - If able to scale down during non-peak to run on average 4 ACUs, monthly cost is $4 * \$0.16 * 24 * 365/12 = \$467.20/\text{month}$
 - Given 1 master + 2 read replicas, will spend $\$467.20/\text{instance} * 3 \text{ instances} = \$1,401.60/\text{month}$
 - For 100k customer scenario, must maintain at least 0.5 ACUs/instance * 3 instances * $\$0.16 * 24 * 365/12 = \175.20 minimum spend
 - Notes on ACUs:
 - Each ACU corresponds to ~2 GiB memory, can have min 0.5 ACUs and increase/decrease in increments of 0.5 ACUs
 - Suppose that we need enough memory to hold all core consent data in memory + ¼ of the audit history in memory, assuming that audit history queries will be presumably less frequent.
 - We've assumed in the "Scaling estimates" section that the audit history data is 4x the size of the consent data, so this brings our total memory to 2x the core consent data.

- For the 1M customer scenario, this gives us 10 GB of total required memory * 1 ACU/2 GiB memory = 5 ACUs required during active usage.
 - Can set minimum and maximum ACUs to automatically scale up/down based on actual usage
- Storage:
 - Standard: \$0.10 per GB-month - 3x for 3 instances if have 1 master + 2 replicas
 - I/O-optimized: \$0.225 per GB-month - ""
- I/O rate:
 - Standard: \$0.20 per 1 million requests
 - I/O-optimized: included
- Global Database
 - Enables replication across multiple global regions with no performance impact, enabling fast local region latency and disaster recovery in case of regional outages
 - Replicated Write I/Os: \$0.20 per million replicated write I/Os
- Point-in-time backups:
 - Base backup charge of \$0.021/GB-month
 - For the 1M customer use case, $\$0.021/\text{GB} * 250 \text{ GB} = \5.25
 - Additional \$0.012 per 1 million cumulative change records covered by point-of-time recovery
 - For the 1M customer example, assume average 5 years of history/customer with average 10 writes/year/customer -> 1M customers * 10 writes/year * 5 years = 50M change records * \$0.012/M change records = \$0.60
 - Total for 1M customer scenario = $\$5.25 + \$0.60 = \$5.85$
- Data transfers out
 - To Internet: Free globally across AWS services for first 100GB, then \$0.09 per GB for first 10 TB
 - To other AWS regions: \$0.02/GB
 - For 10M customers scenario, $10\text{M write transactions/month} * 2 \text{ replications/write} * 1\text{KB/replication} * 1/1024 \text{ MB/KB} * 1/1024 \text{ GB/KB} = 19.07\text{GB} * \$0.02/\text{GB} = \$0.38$
- Ref: <https://aws.amazon.com/rds/aurora/pricing/?pg=pr&loc=1>

Estimated PostgreSQL on RDS costs/month

- Assume multi-AZ deployment with 2 standbys in different regions
- Storage on database: \$0.345 per GB-month, with min 20 GB charge, max 64 TB storage
 - For 250 GB storage on database instance with 3 instances, $\$0.345/\text{GB/month} * 250 \text{ GB/instance} * 3 \text{ instances} = \$258.75/\text{month}$
- IOPs: \$0.06 per IOPS-month over baseline
 - Only pay for periods when above baseline usage, difficult to predict
 - For 1M customer use case, have estimated peak traffic at 250 reads/second, 10 writes/second, assume this is below RDS' baseline supported IOPS
- Throughput: \$0.240 per throughput MB/s-month over baseline

- Backup storage: \$0.095/GB/month
 - Assume backup stores full DB's contents + all writes from the past year
 - For 1M customer use case, 250 GB base DB size + ($\sim 1\text{M transactions/month} * 12 \text{ months/year} * 1\text{KB/write transaction} * 1/1024 \text{ MB/KB} * 1/1024 \text{ GB/KB} \sim 11.5 \text{ GB}$) $\sim 261.5 \text{ GB} * \$0.095/\text{GB/month} \sim \$25/\text{month}$
- Instance hours:
 - Highly dependent on instance type used, many options with different configurations and resource capacities, may not know exactly which instance types to use without load testing against test data
 - Expect to need to scale to larger instances over the course of the first and second year depending on customer usage, so will tend towards using on-demand instances instead of paying up-front for 1-3 years of a fixed instance type, which would otherwise give reserved instance pricing discounts
 - Given we expect to never be able to turn off databases throughout the day, assume 100% utilization with on-demand pricing
 - For the 1M customer use case:
 - Assuming we need $\geq 50 \text{ GB}$ memory, would select the db.m6g.4xlarge general-purpose, Graviton based instance type for 16 vCPUs and 64 GiB memory, lower price than more specialized instance types or Intel based instance types
 - db.m6g.4xlarge pricing: $3 \text{ instances} * \$2.544/\text{hour} * 730 \text{ hours/month} = \$5571.36/\text{month}$
 - For 10M customer use case:
 - Assuming need $\geq 500 \text{ GB}$ memory, db.m6 series no longer supports our needs and need to step up to more expensive memory optimized instance types, might start by load testing against db.r6g.16xlarge which has 64 vCPUs, 512 GiB memory
 - db.r6g.16xlarge pricing: $3 \text{ instances} * \$14.387/\text{hour} * 730 \text{ hours/month} = \$31,507.53$
 - For 100k customer use case:
 - Assuming we need $\geq 5 \text{ GB}$ memory, can use the db.m6g.large instance type, general purpose, Graviton-based instance type with 2 vCPU, 8 GiB
 - db.m6g.large pricing: $3 \text{ instances} * \$0.318/\text{hour} * 730 \text{ hours/month} = \$696.42/\text{month}$
- Data transfers out of region: same as for DynamoDB Global Tables and Aurora Serverless
 - Transfer out from master to 2 replicas in different regions for each write
 - For the 1M customer scenario, $1\text{M writes/month} * 2 \text{ replica data transfers/write} * 1\text{KB/transfer} * 1/1024 \text{ MB/KB} * 1/1024 \text{ GB/MB} \sim 1.9 \text{ GB/month}$
 - Assume all read queries from companies and end users trigger data transfers out, $50\text{M reads/month} * 1 \text{ data transfer/read} * 1\text{KB/transfer} * 1/1024 \text{ MB/KB} * 1/1024 \text{ GB/KB} = 47.68 \text{ GB}$
- RDS performance insights, 1 year retention: \$105.12/month

- RDS Proxy
 - Allows applications to pool and share database connections to improve scalability, establishes connection pool and reuses connections in pool to reduce memory/CPU overhead of new connection for every query.
 - Queues/throttles connections that can't be served immediately, so apps can scale without overwhelming the database.
 - Pricing: $\$0.015/\text{vCPU}/\text{hour} * 730 \text{ hours}/\text{month} * 3 \text{ instances} = \$32.85/\text{vCPU}/\text{month}$
 - For 16 vCPU instance types, $\$32.85/\text{vCPU}/\text{month} * 16 \text{ vCPUs} = \$525.60/\text{month}$
 - For 64 vCPU instance types, $\$32.85/\text{vCPU}/\text{month} * 64 \text{ vCPUs} = \$2,102.40/\text{month}$
 - For 2 vCPU instance types, $\$32.85/\text{vCPU}/\text{month} * 2 \text{ vCPUs} = \$65.70/\text{month}$
 - Ref: <https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/rds-proxy.html>
- Ref: <https://aws.amazon.com/rds/postgresql/pricing/?pg=pr&loc=3>

Consent Management API compute service options

Estimated Lambda costs/month

- On-demand requests: First 1M requests/month free, then $\$0.20/1\text{M requests}$
 - For 10k customer use case:
 - 510k requests/month are within the 1M free requests/month = \$0
 - For 100k customer use case:
 - 5.1M requests/month minus 1M free requests/month = 4.1M paid requests/month * $\$0.20/1\text{M requests} = \0.82
 - For 1M customer use case:
 - 51M requests/month minus 1M free requests/month = 50M paid requests/month * $\$0.20/1\text{M requests} = \10
 - For 10M customer use case:
 - 510M requests/month minus 1M free requests/month = 509M paid requests/month * $\$0.20/1\text{M requests} = \$101.80/\text{month}$
- On-demand compute time: First 400k GB-seconds/month free, then $\$0.0000166667/\text{GB-second}$ for next 6 billion GB-seconds
 - Assume Lambda function executions average 30ms to complete, and Lambda function needs 512 MB of memory
 - For 100k customer use case:
 - $5.1\text{M requests} * 0.03 \text{ seconds/request} * \$0.0000166667/\text{GB-second} * 0.5 \text{ GB} = \1.275
 - For 10k customer use case: \$0.13
 - For 1M customer use case: $10 * 100\text{k customer use case} = \12.75
 - For 10M customer use case: $10 * 1\text{M customer use case} = \127.50
- Provisioned concurrency: can choose to pay for provisioned capacity so can always handle spikes in traffic up to a given concurrency level, not needed if traffic is always < 1k requests/second

- Up to 1M customers, expect ≤ 260 requests/second at peak, no need to provision capacity
- For 10M customers, expect peak traffic of 2.6k requests/second
 - No need for provisioned capacity if expect increases to be gradual
 - If expect sharp spikes up to 2k requests/second that need to handle with zero throttling, where remaining increases can be handled by Lambda, and need to pay for 2000 provisioned concurrent functions to be running full-time, AWS Pricing Calculator gives following monthly costs
 - Provisioned Concurrency charges: $2,000 \text{ concurrency} \times 2,628,000 \text{ seconds} \times 0.50 \text{ GB} \times \$0.0000041667/\text{month} = \$10,950.09/\text{month}$
 - Compute duration charges: $510\text{M requests/month} \times 0.03 \text{ seconds/request} \times 0.5 \text{ GB} \times \$0.0000097222/\text{GB-second} = \$74.37/\text{month}$
 - Provisioned request charges (no first 1M free): $510\text{M requests} \times \$0.0000002/\text{M requests} = \$102/\text{month}$
 - Total: $\$11,126.46/\text{month} \approx \11.1k/month

Ref: <https://aws.amazon.com/lambda/pricing>

Estimated ECS Fargate costs/month

- vCPU costs: $\$0.04048/\text{vCPU/hour}$
 - Can select between 0.25 vCPU and 16 vCPU, where different values support different memory values
- Memory costs: $\$0.004445/\text{GB/hour}$
- Ephemeral storage costs: free for first 20GB, which is more than sufficient for our use case
- Auto-scaling details:
 - We want to minimize throttling and dropped traffic, so we want to always be scaled to have $< 0.1\%$ of traffic dropped during any 5-minute period including daily traffic fluctuations and spikes
 - We can set up auto scaling policies including scaling by CPU utilization, memory utilization, and by schedule, where schedule based autoscaling is useful for regular traffic patterns where we can proactively scale up in advance of when we regularly receive sharp traffic increases that CPU/memory based utilization would be too slow to anticipate
 - Our application is not memory intensive, so we can just scale by CPU utilization (automatically add a task when average fleet CPU utilization $> 80\%$ for 5 minutes in a row) and by schedule (adjust min number of tasks by time of day and day of week)
 - If we need to minimize throttling and have somewhat regular traffic increase and decrease daily cycles with spiky bursts within that traffic shape, assume that we always need to be overscaled to support $20\%+$ higher than the current throughput

- Assume that each task that is already provisioned and running can handle on average 1 call every 20ms including DB query time, for a throughput of 50 calls/second, and that during peak spiky traffic we need to overscale to handle bursts of concentrated concurrency within any given second
- Available discounts:
 - Will not use spot pricing since that is primarily for services that have flexible start/end times and can handle interruptions, while we need consistent, immediate low-latency responses to customer requests
 - May benefit from savings plans (ref: <https://aws.amazon.com/savingsplans/compute-pricing/>) where commit to consistent amount of usage for 1 or 3 year term, larger discount if pay all upfront
 - Plan to use savings plans to pay discounted rate for our average number of running tasks, paid upfront for 1 year, and pay on-demand for scaling above that -> 27% discount for Fargate vCPU and memory costs for average number of running tasks
- For 100k customer scenario:
 - Need to support peak 26 calls/second
 - Assume set up ECS cluster with
 - 1 vCPU and 2 GB memory/task
 - At least 1 task running at all times, autoscaling based on CPU and time of day/week, 2 tasks at peak, with an average number of 1.25 running tasks.
 - Discount multiplier for 27% discount on 1 task, running 1 on-demand task 25% of the time: $((1-0.27)*1 + 1*0.25) \Rightarrow$ equivalent to paying for 0.98 continuously running tasks
 - vCPU costs: $0.98 \text{ tasks/month} * 1 \text{ vCPU/task} * 730 \text{ hours/month} * \$0.04048/\text{vCPU/hour} = 0.98*1*730*0.04048 = \$28.96/\text{month}$
 - Memory costs: $0.98 \text{ tasks/month} * 2 \text{ GB/task} * 730 \text{ hours/month} * \$0.004445/\text{GB/hour} = 0.98*1*730*0.004445 = \$3.18/\text{month}$
 - Total cost: \$32.14/month
- 1M customer scenario:
 - Need to support peak 260 calls/second
 - Assume set up ECS cluster with
 - 1 vCPU and 2 GB memory/task
 - At least 1 task running at all times, 6 tasks at peak, average of 3 running tasks
 - Discount multiplier for 27% discount on 3 tasks, running 3 additional tasks 25% of the time: $((1-0.27)*3 + 3*0.25) \Rightarrow$ equivalent to paying for 2.94 continuously running tasks
 - vCPU costs: $2.94*1*730*0.04048 = \$86.88/\text{month}$
 - Memory costs: $2.94*1*730*0.004445 = \$9.54/\text{month}$
 - Total cost: \$96.42/month
- 10M customer scenario:
 - Need to support peak 2.6k calls/second
 - Assume set up ECS cluster with

- 1 vCPU and 2 GB memory/task
- At least 5 tasks running at all times, 52 tasks running at peak, average of 26 running tasks
- Discount multiplier for 27% discount on 26 tasks, running 26 additional tasks 25% of the time: $((1-0.27)*26 + 26*0.25) \Rightarrow$ equivalent to paying for 25.48 continuously running tasks
- vCPU costs: $25.48*1*730*0.04048 = \$752.94/\text{month}$
- Memory costs: $25.48*1*730*0.004445 = \$82.68/\text{month}$
- Total cost: $\$835.62/\text{month}$

Ref: <https://aws.amazon.com/fargate/pricing/>

Estimated EC2 on ECS costs/month

- For 100k customer scenario
 - Assume select t4g.medium instance type which has 2 vCPUs and 0.5 GB memory
 - On-demand pricing: \$0.0042/hour
 - Reserved pricing: \$22 upfront for 1 year per host
 - Assume can have baseline of 1 task running at all times, with an ECS autoscaling group with scaling policies based on CPU and memory utilization, with average number of 1.5 tasks running at any given time
 - Assume reserve 1 host for 1 year -> \$22 for the year, amortized to \$1.83/month
 - Remaining average 0.5 tasks * 730 hours/month * 0.0042/hour = \$1.53/month
 - Total cost = \$3.36
- 1M customer scenario: 10 * 100k customer scenario, reserve 10 tasks for 1 year with auto-scaling resulting in running 15 total hosts on average, where 5 of those tasks are using on-demand capacity -> \$33.60/month
- 10M customer scenario: \$336.00/month

Ref:

- EC2 Standard pricing: <https://aws.amazon.com/ec2/pricing/>
- EC2 Reserved instances: <https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/>

Consent Management API interface options

Summary price comparison

	Variations	Total estimated monthly costs	API requests	Cache	Metrics
REST API Gateway w/ 100k customers	Cache	\$35.25	\$14.35	\$14.60	\$6.30

	No cache	\$20.65	\$14.35		\$6.30
REST API Gateway w/ 1M customers	Cache	\$209.04	\$175.00	\$27.74	\$6.30
	No cache	\$181.30	\$175.00		\$6.30
REST API Gateway w/ 10M customers	Cache	\$1,847.10	\$1,658.30	\$182.50	\$6.30
	No cache	\$1,664.60	\$1,658.30		\$6.30
HTTP API Gateway w/ 100k customers		\$5.10	\$5.10		
HTTP API Gateway w/ 1M customers		\$51.00	\$51.00		
HTTP API Gateway w/ 10M customers		\$489.00	\$489.00		
Application Load Balancer w/ 100k customers		\$24.31	\$24.31		
Application Load Balancer w/ 1M customers		\$29.57	\$29.57		
Application Load Balancer w/ 10M customers		\$82.13	\$82.13		

Application Load Balancers are by far the least expensive option between a REST API Gateway, HTTP API Gateway, or Application Load Balancer.

REST API Gateways are clearly most valuable from a feature standpoint, but need to decide whether gateway-level auth, per-client rate limiting and throttling, caching, native WAF integration, API gateway-level request validation, and edge-optimized endpoints are worth the price.

For 1M customers, would easily pay the \$209.04/month for the functionality offered by a REST API Gateway, development time saved that would otherwise be spent replicating functionality is worth it.

For 10M customers, slightly more difficult to say whether the REST API Gateway is worth \$1,847.10/month if have limited funding, and this becomes a question of how valuable it is to a company to save developer time for other feature work, or how much the caching offered by REST API Gateway may save on backend hardware costs via reduced compute + database usage.

- Backend hardware savings, assuming caching reduces backend traffic by 20%:

- DynamoDB: Referencing the 10M customer use case from the DynamoDB pricing estimates, this would save $0.2 * (\$125 \text{ read costs} + \$37.50 \text{ write costs} + \$35.62 \text{ data transfer costs}) = \$39.62/\text{month}$ in DynamoDB costs
- EC2 on ECS: This would reduce our required service cluster size by 20%, saving $0.2 * \$336 = \$67.20/\text{month}$
- Total savings of $39.62 + 67.2 = \$106.82/\text{month}$ are negligible compared to the REST API Gateway cost, not a major factor
- Development time savings:
 - If the REST API Gateway functionality the platform would use would take over a month of developer effort to implement with an Application Load Balancer, likely still worth paying for the REST API Gateway.
 - If every deployed instance of the platform were to have only a single client (making per-client usage tracking + rate limiting + throttling unnecessary), in a single region (making edge-optimized endpoints unnecessary), with client traffic coming from just a few well-defined servers that could be covered by stable security group rules rather than anywhere on the Internet (making gateway-level auth less critical), an Application Load Balancer is likely worth the savings. However, if the platform needs non-trivial authentication and authorization logic as well as client rate limiting + throttling, the REST API Gateway is a reasonable price to save that development time.
 - For our platform, if we have a multi-tenancy environment in which multiple clients interact with the same platform, we would want to implement resource-based access policies, per-client usage tracking + rate limiting + throttling, and gateway-level access controls. In this case, recommend the REST API Gateway, and should ensure our pricing model allows for the increased hardware costs to be more than covered by increased revenue from larger clients.

Estimated REST API Gateway/month

- API requests: first 1M/month free (for first 12 months only), then \$3.5/M for next 333M, then \$2.8/M for next 667M
 - For 100k customers, estimated 5.1M requests/month
 - $(5.1\text{M} - 1\text{M free}) \text{ requests/month} * \$3.5/\text{M requests} = 4.1 * 3.5 = \$14.35/\text{month}$
 - For 1M customers, estimated 51M requests/month
 - $(51\text{M} - 1\text{M free}) \text{ requests/month} * \$3.5/\text{M requests} = 50 * 3.5 = \$175/\text{month}$
 - For 10M customers, estimated 510M requests/month
 - $(510\text{M} - 1\text{M free}) \text{ requests/month} = 509\text{M requests/month}$
 - $\text{First } 333\text{M requests/month} * \$3.5/\text{M requests} + (509\text{M} - 333\text{M}) \text{ requests/month} * \$2.8/\text{M requests} = 333 * 3.5 + (509 - 333) * 2.8 = \$1,658.30/\text{month}$
- Caching: pay by cache size
 - Pricing:
 - \$0.02/hour for 0.5 GB
 - \$0.038/hour for 1.6 GB

- \$0.20/hour for 6.1 GB
 - \$0.25/hour for 13.5 GB
 - \$0.50/hour for 28.4 GB
 - \$1/hour for 58.2 GB
- Assume only want to cache consent data, not audit history data
 - For 100k customers:
 - Estimated have 0.5 GB consent data, would use the min cache of 0.5 GB
 - 0.5 GB cache: $\$0.02/\text{hour} \times 730 \text{ hours/month} = \$14.60/\text{month}$
 - For 1M customers:
 - Estimated have 5 GB consent data, and expect no more than 20% of them to have business interactions triggering queries against their consent data at a given time -> 1.6 GB cache option should be sufficient
 - 1.6 GB cache: $\$0.038/\text{hour} \times 730 \text{ hours/month} = \$27.74/\text{month}$
 - For 10M customers:
 - Estimated have 50 GB consent data, only need to cache 20% of that data -> 13.5 GB cache should be sufficient
 - 13.5 GB cache: $\$0.25/\text{hour} \times 730 \text{ hours/month} = \$182.50/\text{month}$
- May not need caching for our use case
 - Will increase customer-facing latency if most requests are cache misses due to distributed traffic and lack of repeated requests
 - Can always add later on if observe high rate of repeated traffic concentrated around small subset of users
- Logs: Negligible cost at all expected scales, not a factor
 - Assume each request generates average of 3 KB access logs + 5 KB execution logs
 - For API Gateway vended CloudWatch Logs, pricing is \$0.5/GB for first TB/month
 - For 100k customers, estimated 5.1M requests/month
 - $5.1\text{M requests/month} \times 8 \text{ KB logs/request} \times 1/1024 \text{ MB/KB} \times 1/1024 \text{ GB/MB} = 5.1 \times 8 / 1024 / 1024 \approx \$0/\text{month}$
 - For 1M customers with 51M requests/month: $51 \times 8 / 1024 / 1024 \approx \$0/\text{month}$
 - For 10M customers with 510M request/month: $510 \times 8 / 1024 / 1024 \approx \$0/\text{month}$
- Metrics:
 - Assume emitting 21 metrics/request from the 7 metrics at <https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-metrics-and-dimensions.html> across 3 groups of dimensions
 - CloudWatch Metrics pricing: \$0.3/metric/month for first 10k metrics - not charging per metric emission, just for each metric that has emissions
 - $21 \text{ metrics} \times \$0.3/\text{month} = \$6.3/\text{month}$

Ref:

- API Gateway pricing: <https://aws.amazon.com/api-gateway/pricing/>
- CloudWatch pricing: <https://aws.amazon.com/cloudwatch/pricing/>

Estimated HTTP API Gateway costs/month

- API requests: \$1/M requests for first 300M requests, then \$0.9/M for subsequent requests
 - For 100k customers w/ 5.1M requests/month: $5.1M * \$1/M = \$5.10/\text{month}$
 - For 1M customers w/ 51M requests/month: $51 * 1 = \$51/\text{month}$
 - For 10M customers w/ 510M request/month: $300 * 1 + 210 * 0.9 = \$489/\text{month}$

Estimated Application Load Balancer costs/month

- \$0.0225 per Application Load Balancer-hour (or partial hour)
 - $\$0.0225/\text{hour} * 730 \text{ hours/month} = \$16.43/\text{month}$
- \$0.008 per LCU-hour (or partial hour)
 - LCUs are measured across 4 dimensions (new connections/second, active connections/minute, processed bytes, rule evaluations where first 10 evaluated rules/hour are free), and for each hour, pay for the dimension with the highest measured LCUs
 - New connections: 1 LCU per 25 new connections/second
 - Active connections: 1 LCU per 3k active connections/minute w/o mutual TLS, or 1 LCU per 1.5k active connections/minute with mutual TLS
 - Processed bytes: 1 LCU per 1 GB/hour for EC2 instances/containers/IPs as targets, or 0.4 GB/hour for Lambda function targets
 - Rule evaluations: 1 LCU per 1k rule evaluations/second
- \$0.005/hour/associated trust store when using Mutual TLS (or partial hour)
 - Assume use up to 2 associated trust stores, eg 1 AWS-managed trust store + 1 company-provided trust store
 - $\$0.005/\text{hour/trust store} * 2 \text{ trust stores} * 730 \text{ hours/month} = \$7.30/\text{month}$
- Scale:
 - For 1M customers, scale estimates gave
 - New connections:
 - End-customer-based traffic of 1.833M requests/month with peak traffic of 11 requests/second, which gives ≤ 11 new connections/second
 - Company-based traffic of up to 233 requests/second at peak, where these requests could largely be made through existing connections, assume 10% are new connections
 - Guesstimate ~ 25 new connections/second $\rightarrow 1$ LCU
 - Action connections:
 - Average traffic of $51M \text{ requests/month} * 1/730 \text{ months/hour} * 1/3600 \text{ hours/second} = 19.406 \text{ requests/second} * 60 \text{ seconds/minute} * 1 \text{ LCU}/1.5k \text{ active connections/minute w/ mutual TLS} = 19.406 * 60 / 1500 < 1 \text{ LCU}$
 - Processed GB for Lambda function target: $51M \text{ requests/month} * \text{average of } 5 \text{ KB processed by load balancer per request-response pair} * 1/1024 \text{ MB/KB} * 1/1024 \text{ GB/MB} = 51,000,000 * 5 / 1024 / 1024 = 243.19 \text{ processed}$

- Expect average Consent History item is ~3 KB (equivalent to 384 8-byte characters in payload), so this reduces to \$0.10/M data capture units * 3 data capture units/consent update = \$0.30/consent update
- Have estimated that N customers on average make N consent updates/month
- For 100k customers:
 - 100k customers * 1 consent update/customer/month * \$0.30/M consent updates = 0.1M consent updates/month * \$0.30/M consent updates = \$0.03/month
- For 1M customers: $1 * 0.3 = \$0.30/\text{month}$
- For 10M customers: $10 * 0.3 = \$3/\text{month}$
- Per-stream-per-hour costs:
 - \$0.04/hour/stream * 1 data stream * 730 hours/month = \$29.20/month
- Data ingested, per GB (Includes 24-hour retention):
 - \$0.08/GB ingested
 - Expect to ingest N Consent History items/month for N customers, with each item averaging 3KB
 - For 100k customers:
 - 100k customers * 1 Consent History item/month/customer * 3KB/Consent History item * 1/1024 MB/KB * 1/1024 GB/MB * \$0.08/GB = $100,000 * 3 / 1024 / 1024 * 0.08 \approx \$0.02/\text{month}$
 - For 1M customers: $1,000,000 * 3 / 1024 / 1024 * 0.08 \approx \$0.23/\text{month}$
 - For 10M customers: $10,000,000 * 3 / 1024 / 1024 * 0.08 \approx \$2.29/\text{month}$
- Data retrievals, per GB
 - \$0.04/GB retrieved
 - For 100k customers: $100,000 * 3 / 1024 / 1024 * 0.04 \approx \$0.01/\text{month}$
 - For 1M customers: $\approx \$0.12/\text{month}$
 - For 10M customers: $\approx \$1.15/\text{month}$
- Enhanced fan-out data retrievals, per GB
 - \$0.05/GB retrieved
 - For 100k customers: $100,000 * 3 / 1024 / 1024 * 0.05 \approx \$0.01/\text{month}$
 - For 1M customers: \$0.14/month
 - For 10M customers: \$1.43/month
- Data storage:
 - Free for 24 hour data retention
 - \$0.10/GB-month beyond 24 hours, up to 7 days
 - Accepting 24 hour data retention as sufficient given use of DLQ
- Total costs:
 - 100k customers: $0.03 + 292 + 0.02 + 0.01 + 0.01 = \$292.07/\text{month}$
 - 1M customers: $0.3 + 292 + 0.23 + 0.12 + 0.14 = \$292.79/\text{month}$
 - 10M customers: $3 + 292 + 2.29 + 1.15 + 1.43 = \$299.87/\text{month}$

Ref:

- Data capture pricing for DynamoDB -> Kinesis Data Streams:
<https://aws.amazon.com/blogs/database/choose-the-right-change-data-capture-strategy-for-your-amazon-dynamodb-applications/>
- Base Kinesis Data Stream pricing: <https://aws.amazon.com/kinesis/data-streams/pricing/>

Estimated Lambda function costs/month

- SQS costs are equivalent between the data stream options.
- On-demand requests: First 1M requests/month free, then \$0.20/1M requests
 - For 100k customer use case: \$0/month
 - For 1M customer use case: \$0/month
 - $1\text{M Consent History events/month} * 1.05 \text{ average requests/Consent History event given redrives with } 0.05\% \text{ failure rate} = (1.05\text{M requests/month} - 1\text{M free requests/month}) * \$0.20/1\text{M paid requests} = 0.05 * 0.2 = \$0.01/\text{month}$
 - For 10M customer use case:
 - $(10 * 1.05 - 1) * 0.2 = \$1.90/\text{month}$
- On-demand compute time: First 400k GB-seconds/month free, then \$0.0000166667/GB-second for next 6 billion GB-seconds
 - Assume Lambda function executions average 30ms to complete, and Lambda function needs 512 MB of memory
 - For 100k customer use case:
 - $1.05\text{M requests} * 0.03 \text{ seconds/request} * \$0.0000166667/\text{GB-second} * 0.5 \text{ GB} = 1.05 * 0.03 * 0.0000166667 * 0.5 \approx \$0/\text{month}$
 - For 1M customer use case: $\approx \$0/\text{month}$
 - For 10M customer use case: $\approx \$0/\text{month}$
- Provisioned concurrency: can choose to pay for provisioned capacity so can always handle spikes in traffic up to a given concurrency level, not needed if traffic is always < 1k requests/second
 - For 10M customers, expect peak traffic of 100 Consent History events/second * 1.05 call ratio to account for failure retrieves = 105 requests/second at peak < 1k requests/second, no need to provision capacity
- Total cost: \$0/month for up to 10M customers

Ref: <https://aws.amazon.com/lambda/pricing>

Estimated SQS costs/month

- SQS costs are equivalent between the data stream options, where we only expect costs from DLQ items when there are Consent History Ingestor Lambda failures.
- SQS does not charge for the number of messages in the queue, and does not charge for data transfers in or data transfers from SQS to a Lambda function in the same AWS region
- Per request charges differ between Standard and FIFO queues with FIFO queues charging more, however, we do not need FIFO queues for the data streaming options

where every consent data update includes its relevant versioning information, so we can go with standard SQS queues

- Standard SQS queue requests charge:
 - \$0/month for the first 1M requests/month
 - \$0.40/month for request from 1M-100B/month
- Estimating scale:
 - Estimated that N end customers will trigger average N consent updates/month
 - Every consent update will trigger 1 Consent History event
 - SQS messages will only be triggered if the Consent History Ingester Lambda function fails to process an incoming event
 - Assume average failure rate of 0.05% (5 in 10k events failing processing), based on Lambda availability SLA of 99.95% and sparse server-side error edge cases that need to be investigated and resolved
 - Assume the average failed event triggers 1 SQS SendMessage request (pushing event to queue) and 2 SQS ReceiveMessage requests (Lambda polling queue to attempt processing), to account for on-call remediate + redrive efforts -> 3 SQS requests/Consent History event processing failure
 - Assume baseline SQS polling of once per 30 seconds regardless of number of messages in the queue, with failed processing adding event based requests on top of this -> baseline of 1 request/30 seconds * 3600 seconds/hour * 24 hours/day * 365/12 days/month = 87.6k requests/month
- Expected pricing based on number of customers:
 - For 100k customers: \$0/month
 - $100k \text{ customers} * 1 \text{ consent update/month/customer} * 3 \text{ SQS requests/consent update} + \text{baseline } 87.6k \text{ SQS requests/month} = 387.6k \text{ SQS requests/month}$
 - Within free tier, \$0/month
 - For 1M customers:
 - $1M * 1 * 3 + 87.6k = 3.0876M \text{ requests/month} - 1M \text{ free requests/month} = 2.0876M \text{ paid requests/month} * \$0.40/\text{M requests/month} = 2.0876 * 0.4 \approx \$0.84/\text{month}$
 - For 10M customers:
 - $10M * 1 * 3 + 87.6k = (30.0876M \text{ requests/month} - 1M \text{ free requests/month}) * \$0.40/\text{M requests/month} = (30.0876 - 1) * 0.4 \approx \$11.64/\text{month}$

<https://aws.amazon.com/sqs/pricing/>

Security service options

Estimated AWS Web Application Firewall (WAF) costs/month

Note: WAF is the most important security-specific AWS service considered here, recommended to protect any Internet-facing APIs and web applications.

- Web ACL: \$5/month/ACL
 - Assume have one ACL -> \$5/month
- Rule charges: \$1/month/rule
 - Assume apply 10 rules -> \$1/month/rule * 10 rules = \$10/month
- Request charges: \$0.60 per 1 million requests
 - For 100k customer scenario: expect valid traffic of 5.1M requests/month * \$0.6/M requests = \$3.06/month
 - If get additional 20% invalid/bot traffic -> $3.06 * 1.2 = \$3.67/\text{month}$
 - For 10k customer scenario: \$0.37
 - For 1M customer scenario: $51 * 0.6 = \$30.60/\text{month}$
 - With additional 20% invalid/bot traffic -> \$36.70/month
 - For 10M customer scenario: $510 * 0.6 = \$306/\text{month}$
 - With additional 20% invalid/bot traffic -> \$367/month
- Rule actions:
 - Allow/block/count are free
 - If choose to apply captcha challenges for some rules, pay for the number of challenges, and the number of completed captcha attempts:
 - Challenges: \$0.4/M challenges issued
 - Captcha completions analyzed: \$4/10k completed captchas analyzed
- If choose to enable Bot Control rules, can scope down which rules should result in bot analysis and captcha challenges if meet bot criteria:
 - Common Bot Controls: \$1/M requests inspected, challenge/captcha result analysis costs as above
 - For 1M customer scenario + 20% invalid/bot traffic: 51M valid requests * $1.2 * \$1/\text{M} = 51 * 1.2 * 1 = \$61/\text{month}$
 - For 10M customer scenario + 20% invalid/bot traffic: \$610/month
 - Appears easy to bypass, likely not worth it unless start observing large amount of bot traffic, start with basic WAF rules which are less expensive and already support rate limiting
 - Targeted Bot Controls: \$10/M requests inspected, challenge/captcha result analysis are included - additional functionality here to catch non-self-identifying bots
 - For 1M customer scenario + 20% invalid/bot traffic: 51M valid requests * $1.2 * \$10/\text{M} = 51 * 1.2 * 10 = \$610/\text{month}$
 - For 10M customer scenario + 20% invalid/bot traffic: \$6.1k/month
 - More likely to catch bots than Common Bot Controls, however, seems unlikely to be worth it compared to simple WAF rules which include rate limiting rules.
 - Start without this, can re-evaluate if start seeing large amount of bot traffic that cannot be mitigated via basic WAF rules
- Total costs assuming 20% invalid/bot traffic on top of legitimate traffic:
 - 10k customers scenario: $5 + 10 + 0.37 = \$15.37/\text{month}$
 - 100k customers scenario: $5 + 10 + 3.67 = \$18.67/\text{month}$

- 1M customers scenario: $5+10+36.7 = \$51.70/\text{month}$
- 10M customers scenario: $5+10+367 = \$382/\text{month}$

Ref: <https://aws.amazon.com/waf/pricing/>

Estimated AWS Shield costs/month

AWS Shield provides DDOS protections for Internet-facing web applications.

- AWS Shield Standard is provided for free for all AWS customers, protects against most common network and transport-layer DDOS attacks.
- AWS Shield Advanced is a more extensive paid service that requires a 1-year subscription commitment, and charges \$3k/month

Does not look necessary to pay for AWS Shield Advanced as a small start-up, will start with basic WAF rules and the free AWS Shield Standard, and re-evaluate if/when large enough for Advanced to be worth the cost.

Total estimated costs based on selection of AWS Shield Standard: \$0/month

Ref: <https://aws.amazon.com/shield/pricing/>

Foundational services

Estimated CloudWatch costs/month

- Dashboards: \$9/month
 - \$3/month/custom dashboard * 3 custom dashboards (business metrics summary, operational dashboard AKA details, and alarms) = \$9/month
- Metrics: \$3/month
 - First 10 custom metrics free, then \$0.30/month/custom metric for next 10k custom metrics
 - Assume 20 custom metrics
 - $(20 \text{ custom metrics} - 10 \text{ free custom metrics}) * \$0.30/\text{month/custom metric} = (20-10)*0.3 = \$3/\text{month}$
 - First 1M API requests/month free
- Alarms: \$2.50/month
 - First 10 alarm metrics free, then \$0.10/alarm/month for standard metrics (time period ≥ 60 seconds), or \$0.50/compositeAlarm/month for composite alarms
 - Assume 20 standard alarm metrics and 3 composite alarm metrics
 - Standard alarm costs: $(20 \text{ alarm metrics} - 10 \text{ free alarm metrics}) * \$0.10/\text{alarm} = (20-10)*0.1 = \$1/\text{month}$
 - Composite alarm costs: $3 \text{ composite alarms} * \$0.50/\text{compositeAlarm/month} = \$1.50/\text{month}$
- Logs: \$0.02/month
 - Data storage: first 5 GB free, then \$0.50/GB for standard storage

- Assume for first year remain within free tier -> \$0/month
- CloudWatch Log Insights: \$0.005 per GB of data scanned
 - Assume average Log Insights queries run against 100 MB worth of log data, and run average of 50 such queries/month -> 50 queries/month * 100 MB/query * 1/1024 GB/MB * \$0.005/GB data scanned =
 $50 * 100 / 1024 * 0.005 \approx \$0.02/\text{month}$
- Total: $9 + 3 + 2.5 + 0.02 = \$14.52/\text{month}$

Ref: <https://aws.amazon.com/cloudwatch/pricing/>