Добро пожаловать в

Serverless

О чем поговорим?

- Вводная часть: Serverless – что за зверь и зачем он нужен?
- Основные проблемы и стандартные пути решения
- Почему Serverless и Микросервисы крепко дружат?
- Архитектура на Serverless в продакшене

Мой бэкграунд

- В разработке больше 10 лет, нынче Solution Architect
  - Увеличиваю прибыль бизнеса через IT инструменты
- Три продакшена на serverless, плюс пет-проекты
  - Один разработан с нуля и развивался 4 года
  - Другой – поддержка и развитие готового прода
  - Третий – проект с нуля на имеющийся инфраструктуре

Эволюция хостинга

IAAS: ВИРТУАЛЬНЫЕ МАШИНЫ

PAAS: MANAGED KUBERNETES

SAAS: SERVERLESS FUNCTIONS

Serverless как концепция

Wiki:
Бессерверные вычисления (англ. *serverless computing*) — стратегия организации платформенных облачных услуг, при которой облако автоматически и динамически управляет выделением вычислительных ресурсов в зависимости от пользовательской нагрузки.

Мы поговорим в первую очередь о **Serverless Functions –** то есть, об услуге serverless-хостинга вашего кода.

Что предлагает рынок?

AWS: Lambda (с 2014, на .NET с 2016)

Azure: Functions

Google Cloud Platform: Cloud Functions

… многие другие

Включая Yandex.Cloud, Cloud.ru (SberCloud), прочие

Serverless функция

- Абстракция
- «Микро-контейнер» с вашей сборкой
  - Выбираем RAM/CPU/Network
  - Выбираем .NET Runtime
- Не привязана к серверу, к количеству запущенных инстансов
- Вызывается определенная DLL -> Class -> Method
- Треггиры: HTTP, Kafka*, Vendor-specific
- Отработала запрос, вернула ответ
- Нет запросов – «уснула»
- Платим только за время использования

AWS Lambda in .NET

```
public class AwsFunction
{
    public async Task<APIGatewayProxyResponse> Handle(
        APIGatewayProxyRequest request,
        ILambdaContext context)
    {
        return new APIGatewayProxyResponse() { StatusCode = 204 };
    }
}
```

Yandex.Function in .NET

```
public class YandexFunction : YcFunction<Request, Task<Response>>
{
    // Request and Response here - hand-written classes
    public async Task<Response> FunctionHandler(Request request, Context context)
    {
        return new Response() { StatusCode = 204 };
    }
}
```

Микросервисы из коробки

- Разделение ответственности: одна функция – один endpoint/триггер
- Stateless* как пусть к масштабированию и отказоустойчивости
  - Отказоустойчивость: сервис всегда запущен
  - Масштабирование за секунды

Микросервисы из коробки

Минимизация синхронного взаимодействия

Ограничение работы синхронных вызовов

Логирование и мониторинг

Формально не привязаны к стеку разработки, HTTP/Messaging

Можно отдельно тюнить железо каждого сервиса

Serverless + Microservices = Love

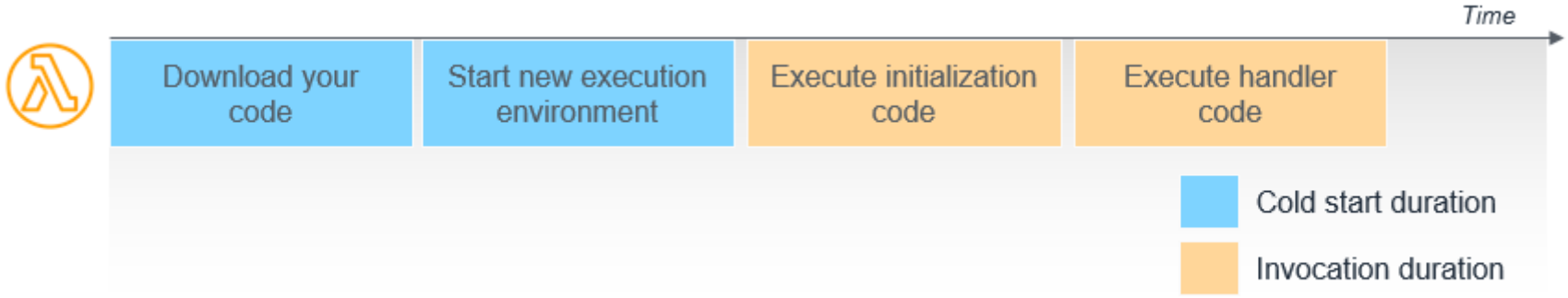
Можно использовать для создания ботов и прочих мелочей.

Level 2: выносим функционал с неравномерной нагрузкой, например, генерация отчетов

Level 80: Полностью строим микросервисную архитектуру на функциях

Проблема: Cold Start

Когда поднимается новый инстанс, инициализация занимает время (до нескольких секунд).



Download your code = Загрузка наших DLL

Execute initialization code = JIT + инициализация до вызова

https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html

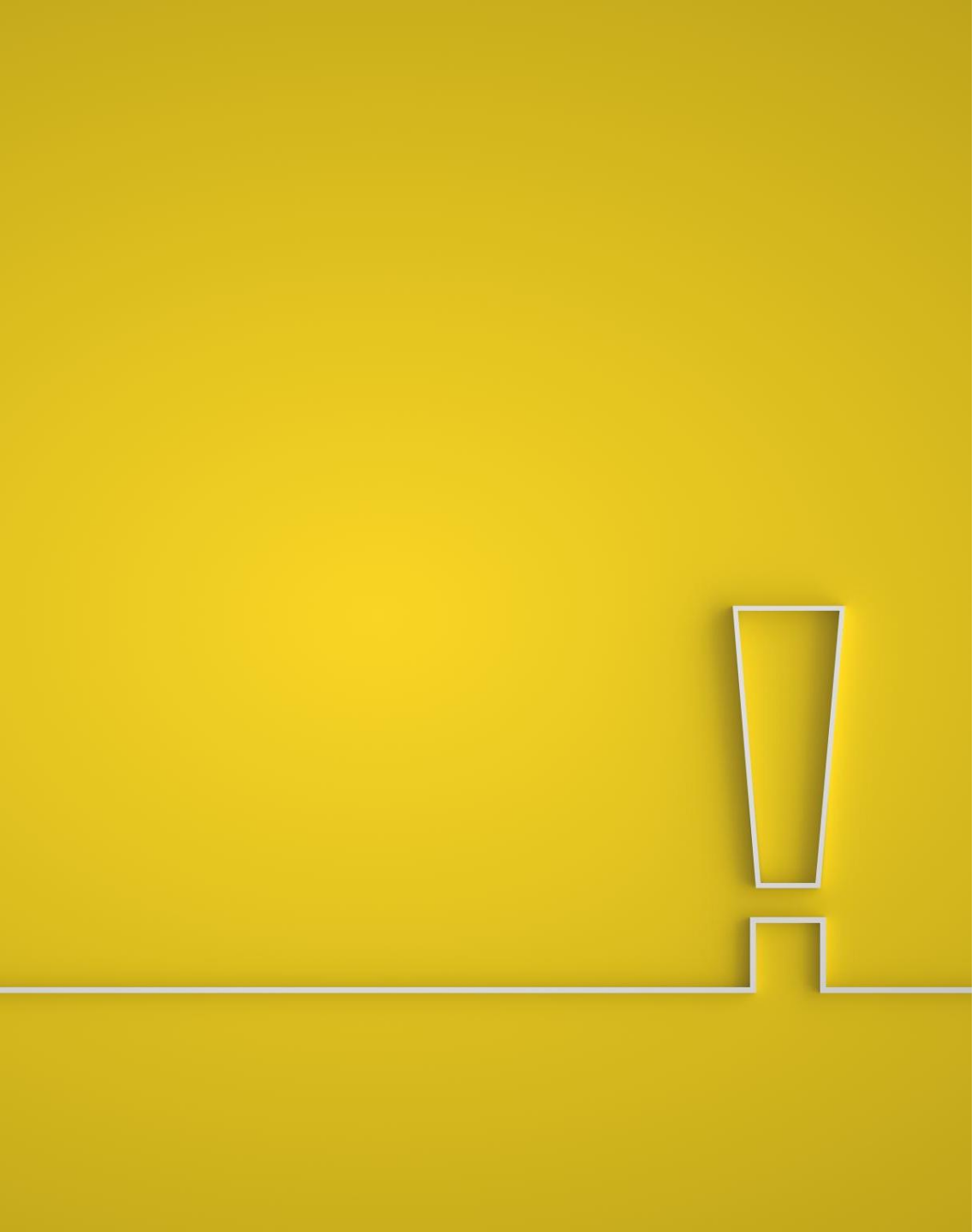
Решение: Cold Start

- Минимизируем размер сборки
  - .NET AWS SDK немного помогает
  - Говорят*, AOT compilation снижает время cold start на <= 90%!
- Минимизируем инициализацию
- Авто-пинг (test environments)
- «Always Warm» фичи от вендора

* https://blog.martincostello.com/native-aot-make-dotnet-lambda-go-brr/

Проблема: Vendor Lock

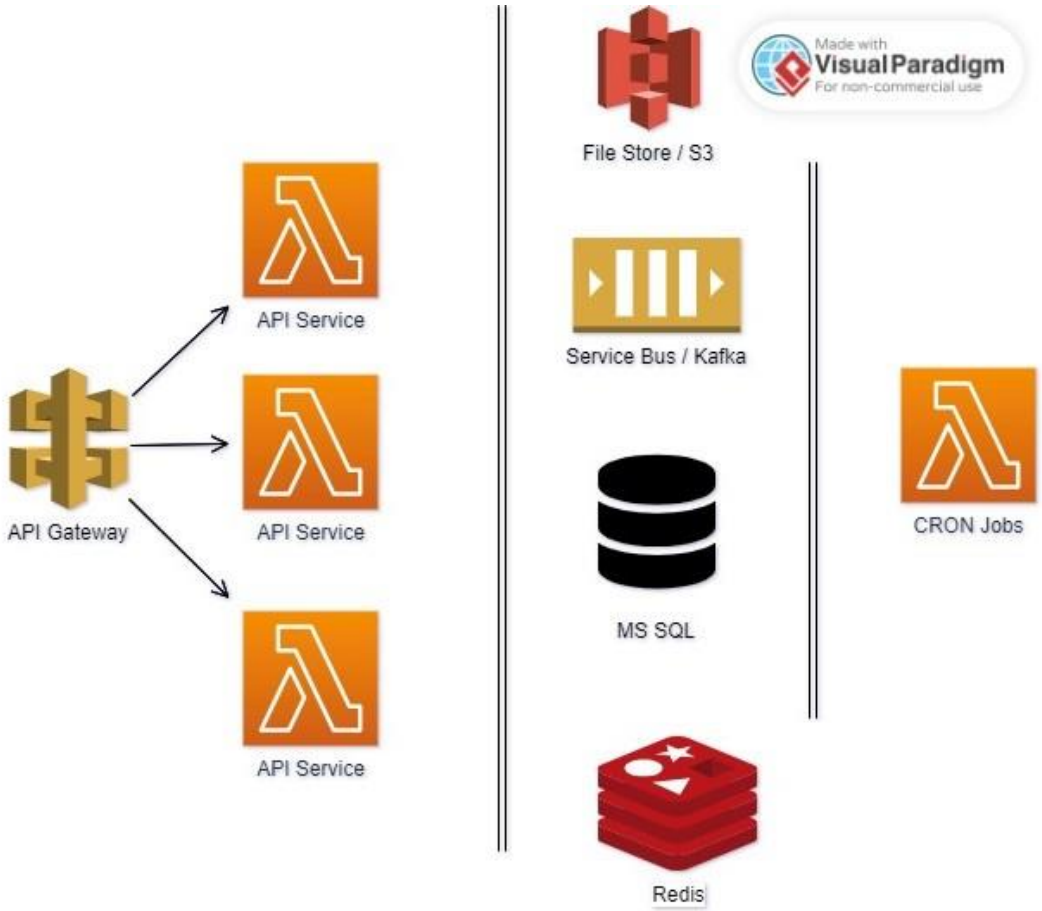
- Сама функция – в большой степени Vendor agnostic
- Можно засунуть ASP.NET Core внутрь функции (но – ColdStart)
- Все триггеры, кроме HTTP – vendor-specific*
  - Интеграция Event Bus соблазнительна
  - Еще более интересны триггеры хранилищ – файлового, БД



Прочие нюансы

- Задачи по таймеру – отдельный сервис
- Ограничение времени выполнения
  - Привет, правильная огранизация фоновых задач и хороший UX
- Это не контейнер, подключиться нельзя
- Никаких длительных соединений (прощай, web-sockets*)
- Никаких фоновых Tasks (заморозка)

Классическая архитектура (AWS)



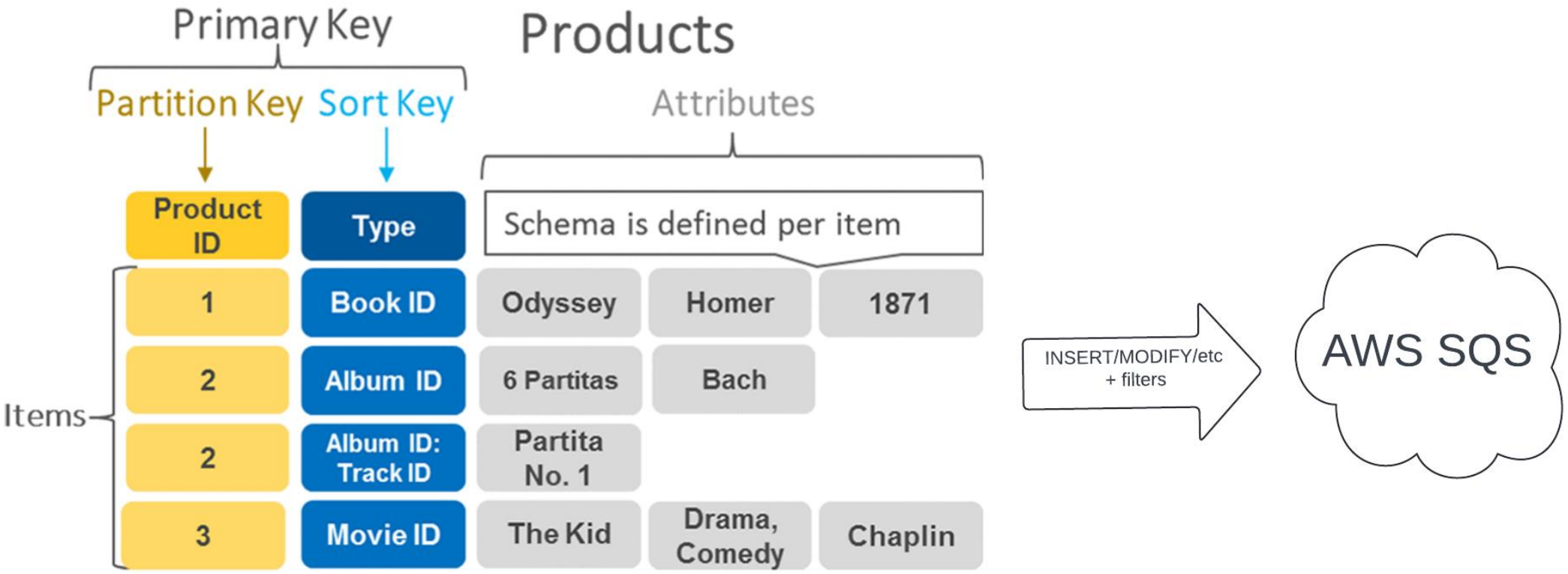
Serverless Функция + InMemory Cache

- Функция как правило переиспользуется
  - По опыту: от нескольких минут до часа
- Внутреннее состояние сохраняется
- Можно использовать
  - Singleton (если у вас DI)
  - Static data (например, MemoryCache)
  - File system

Serverless базы данных: DynamoDb

- Key-Value NoSQL база
- Каждый ключ – сомбинация Partition Key и Sort Key
- Документы с общим Partition Key можно сортировать по SortKey и быстро выгружать с поддержкой пагинации
- Есть Dynamo Streams: любые изменения сущностей могут генерировать события в Event Bus (Trigger Lambda -> AWS SQS)

Serverless базы данных: DynamoDb

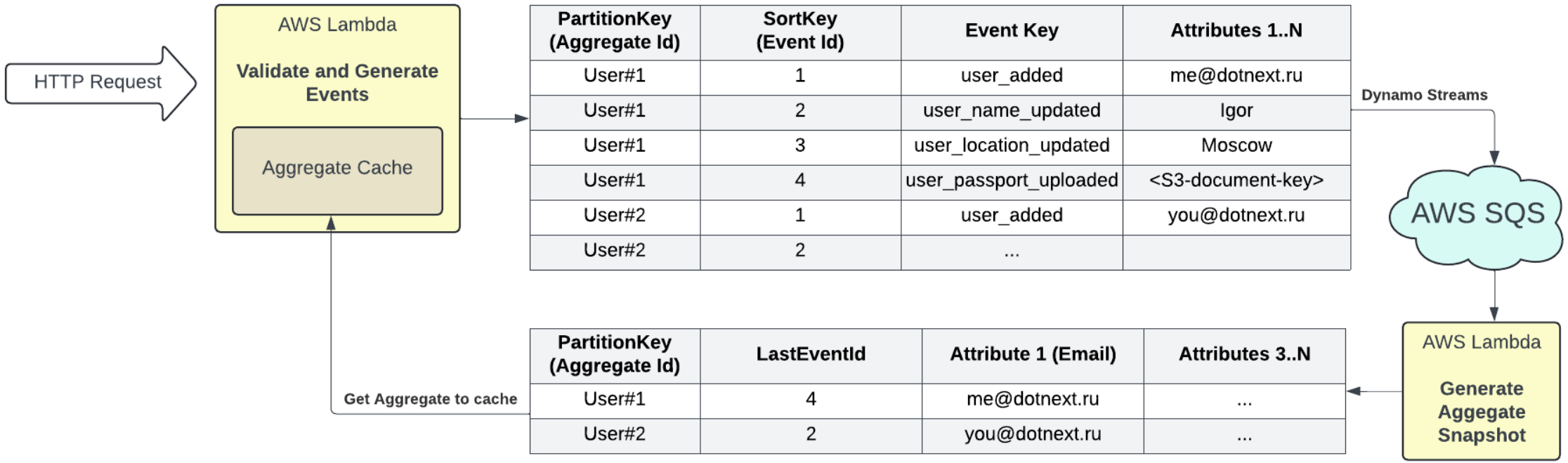


https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/

Продвинутый уровень: CQRS + Event Sourcing

- Функции четко разделены по доменам
- Каждая функция может писать события в DynamoDb
- События сгруппированы по Partition Key – ключу агрегата, и могут быть загружены по Sort Key
- Агрегат можно кэшировать на уровне AWS Lambda, догружая события из базы по мере необходимости
  - Snapshot никто не отменял

Продвинутый уровень: CQRS + Event Sourcing

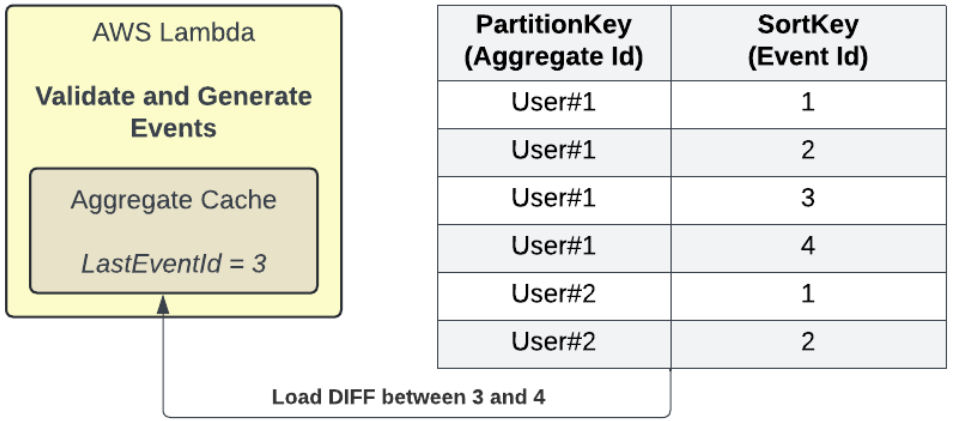


PartitionKey (Aggregate Id)	SortKey (Event Id)	Event Key	Attributes 1..N
User#1	1	user_added	me@dotnext.ru
User#1	2	user_name_updated	Igor
User#1	3	user_location_updated	Moscow
User#1	4	user_passport_uploaded	<S3-document-key>
User#2	1	user_added	you@dotnext.ru
User#2	2	...	

PartitionKey (Aggregate Id)	LastEventId	Attribute 1 (Email)	Attributes 3..N
User#1	4	me@dotnext.ru	...
User#2	2	you@dotnext.ru	...

CQRS: Актуальное состояние агрегата

- Имеем *некоторую* версию состояния агрегата (кэш) **в памяти функции**
- *Select * Where*
  - *PartitionKey = User#1*
  - *SortKey > 3*
- Итог: дешево имеем актуальное состояние
- Знаем id последнего эвента
- Обновляем кэш



PartitionKey (Aggregate Id)	SortKey (Event Id)
User#1	1
User#1	2
User#1	3
User#1	4
User#2	1
User#2	2

CQRS: Актуальное состояние агрегата

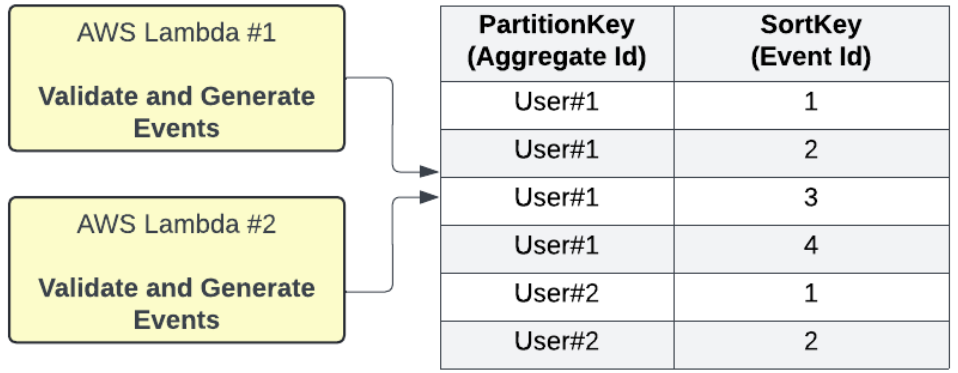
```
public class ViewManager
{
    private readonly IEventsRepository _eventsRepository = new EventsRepository();
    private readonly IUsersRepository _usersRepository = new UsersRepository();
    private readonly ConcurrentDictionary<string, UserAggregate> _cache = new();

    public UserAggregate GetUser(string userId)
    {
        // 1. Get aggregate from cache/database
        var user = _cache.TryGetValue(userId, out var cachedUser)
            ? cachedUser
            : _usersRepository.GetUser(userId);

        // 2. Generate up-to-date state by loading new events
        var newEvents = _eventsRepository.GetEvents(userId, offsetEventId: user.LastEventId);
        return _cache[userId] = user.ApplyEvents(newEvents);
    }
}
```

CQRS: Актуальное состояние агрегата

```
public record UserAggregate(long LastEventId /* More properties here */)
{
    public UserAggregate ApplyEvents(IEnumerable<IEvent> events)
    {
        return events.Aggregate(this, (state, ev) =>
        {
            // TODO: update state based on event
            return state with { LastEventId = ev.Id };
        });
    }
}
```

CQRS: Конкурентная запись событий

- Поскольку есть Aggregate.LastEventId, знаем Id новых эвентов
- База гарантирует уникальность ключа
- Нельзя изменять агрегат параллельно – **не баг, а фича**
- Можно делать retry с обновленным агрегатом



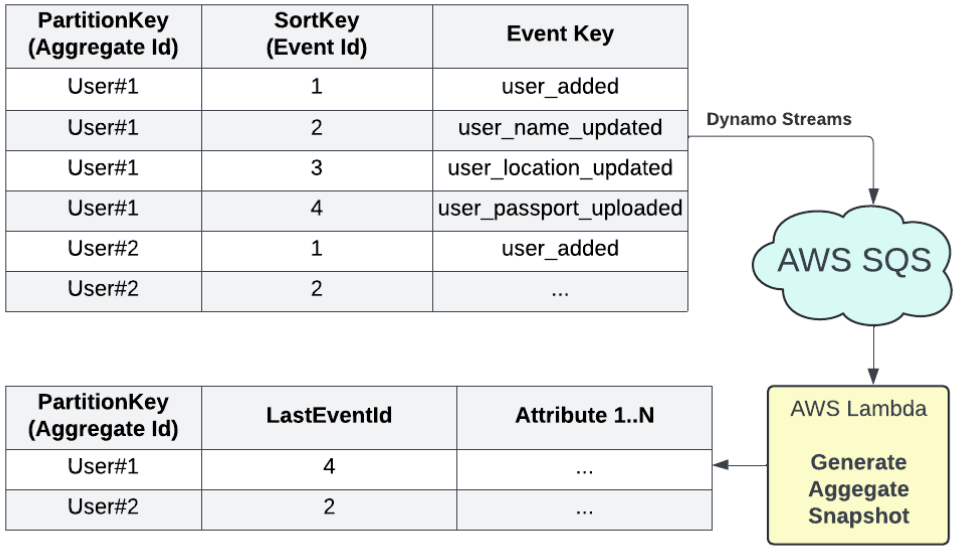
PartitionKey (Aggregate Id)	SortKey (Event Id)
User#1	1
User#1	2
User#1	3
User#1	4
User#2	1
User#2	2

CQRS: Конкурентная запись событий

```
private readonly IDynamoClient _dynamoClient = new DynamoClient();

public Task SaveEvents(IEnumerable<IEvent> events)
{
   var upserts = events.Select(ev =>
   {
      var dbEntity = ev.ToDynamo(); // Has key PK = { PartitionKey: "User#111“, SortKey: "1" }
      return new TransactionUpsertOperation(
         dbEntity,
         "attribute_not_exists(#PK)",
         new Dictionary<string, string>() { ["#PK"] = "PK" });
   });
   return _dynamoClient.WriteTransaction(upserts);
}
```

CQRS: Запись snapshots

1. Загрузить старый снапшот
2. Применить эвенты
3. Сохранить обновленный снапшот
    1. Upsert Where LastEventId <= snapshot.LastEventId
    2. Ингорим фэйлы – кто-то постарался за нас



CQRS: Запись snapshots

```
private readonly IDynamoClient _dynamoClient = new DynamoClient();

/// <inheritdoc />
public Task SaveUser(UserAggregate user)
{
    var dynamoEntity = user.ToDynamo(); // Has key PK = { PartitionKey: "User#111“, SortKey: "1" }
    return _dynamoClient.Upsert(
        dynamoEntity,
        "attribute_not_exists(#LastEventId) or #LastEventId < :LastEventId",
        new Dictionary<string, string>() { [":LastEventId"] = dynamoEntity.LastEventId.ToString() }
    );
}
```

CQRS: Immutable Records

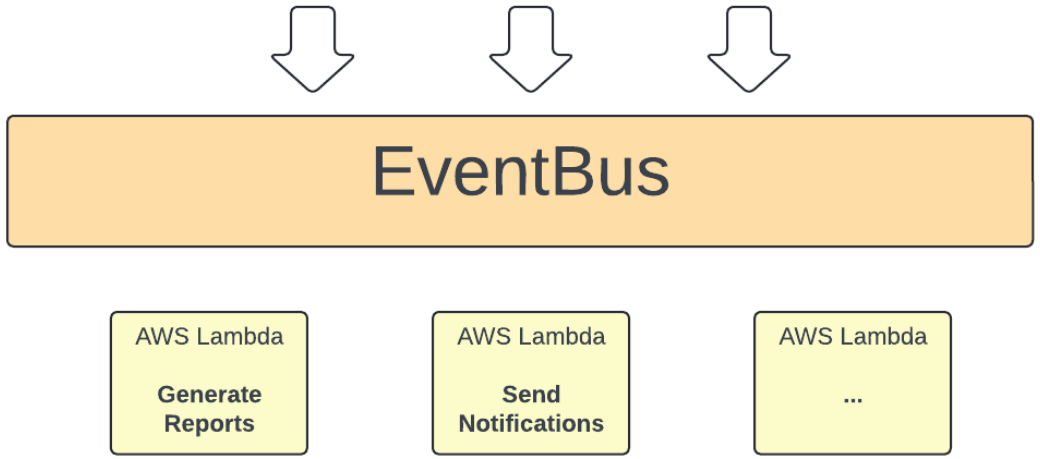
- Эвенты immutable
  - Аудит из коробки
- Снапшоты immutable
- Можно кэшировать где угодно и как угодно!
  - Устраивает eventual consistency – просто кэшировать и читать агрегаты
  - Redis для агрегатов
  - Всегда дешево получить актуальное состояние

PartitionKey (Aggregate Id)	SortKey (Event Id)	Event Key	Attributes 1..N
User#1	1	user_added	me@dotnext.ru
User#1	2	user_name_updated	Igor
User#1	3	user_location_updated	Moscow
User#1	4	user_passport_uploaded	<S3-document-key>
User#2	1	user_added	you@dotnext.ru
User#2	2	...	

PartitionKey (Aggregate Id)	LastEventId	Attribute 1 (Email)	Attributes 3..N
User#1	4	me@dotnext.ru	...
User#2	2	you@dotnext.ru	...

EventBus и доменные события

PartitionKey (Aggregate Id)	SortKey (Event Id)	Event Key	Attributes 1..N
User#1	1	user_added	me@dotnext.ru
User#1	2	user_name_updated	Igor
User#1	3	user_location_updated	Moscow
User#1	4	user_passport_uploaded	<S3-document-key>
User#2	1	user_added	you@dotnext.ru
User#2	2	...	



Логирование

- Как правило, из коробки в лог систему провайдера
  - Azure: Application Insights
  - AWS: CloudWatch
  - Яндекс.Облако: Cloud Logging
- Структурированное логирование
- Можно прямо в консоль
  - Но часто есть поддержка ILogger из коробки

Мониторинг

- Основные метрики из коробки. Как правило:
  - Вызовы (всех или конкретной функции)
  - Длительность работы
  - Количество ошибок
  - Memory/CPU/другое

Принцип оплаты

- Стоимость функций прямо пропорциональна
  - Количеству вызовов
    - 0.2$ / 1kk (Azure/AWS) – примерно 0.4 rpc стабильно весь месяц
  - Времени выполнения
    - Cold-starts учитываются
  - Выделенному железу
    - Оперативная память
    - CPU
    - Пропускная способность сети

Выводы

Serverless хорош:

- Не игрушка, а хороший фундамент для боевых систем
- Экономит ресурсы разработки для бизнес задач
- Прямой путь к масштабируемой микросервисной архитектуре

Нюансы:

- Не про хардкорный HighLoad
- Имеет свои ограничения (см. часть про распространенные проблемы)

Life-hack: Перенос синхронных вызовов в фоновые задачи

- Лямбда получает запрос. Может выдать ответ сразу, если это быстро
- Если не сразу – генерирует Id задачи и отдает клиенту Redirect с <request_id>
  - Сохраняет весь HTTP запрос (заголовки, тело) в базу
  - При сохранении ставим задачу в очередь
- Читаем сообщения из очереди, и сохраняем в базе заголовки и тело ответа
- При HTTP запросе по <request_id> смотрим в базе, если есть ответ – выставляем заголовки и возвращает тело ответа

Life-hack: Code Sample

```
public async Task<APIGatewayProxyResponse> HandleHttp(APIGatewayProxyRequest request)
{
   if (request.QueryStringParameters.TryGetValue("requestId", out var requestId))
   {
      var response = await _requestRespository.GetResponse(requestId);
      return response ??
           Redirect(request.Path + $"?requestId={requestId}");
   }

   // This will trigger SQS event via DynamoStreams
   requestId = await _requestRespository.SaveRequest(request);

   return Redirect(request.Path + $"?requestId={requestId}");
}

public async Task<APIGatewayProxyResponse> HandleSqs(SQSEvent sqsEvent)
{
   var response = YourCustomCode.Process(sqsEvent.ExtractRequest());
   await _requestRespository.SaveResponse(response);
}
```

Спасибо за внимание