Stateful serverless application for data pipeline processing

Stefano Belli
Università degli Studi di Roma Tor Vergata
Macroarea di Ingegneria
Roma, Lazio, Italia
stefano9913@gmail.com

ABSTRACT

Con il seguente documento si intende discutere della realizzazione, e poi del deployment su AWS (Amazon Web Services), di una pipeline per preprocessing di dati critici, sfruttando il pattern serverless SAGA. Vengono discusse scelte progettuali e di implementazione della soluzione. Una tale pipeline è utile nel caso in cui i dati trattati siano di tipo sanitario o comunque di natura critica. Per adempire a task di questo genere (ovvero, di breve durata, semplici e coincisi) è possibile e spesso consigliato ricorrere al serverless. Essendo critici, i dati, quando non corretti secondo una delle fasi di preprocessing (rappresentate da una funzione serverless), vanno comunque preservati. Altrimenti, dopo essere stati preprocessati correttamente, saranno inseriti in un apposito storage consultabile da un client/consumer. Si decide di progettare e poi implementare il preprocessing secondo pattern SAGA - in questo caso - *"serverless SAGA"*.

1 INTRODUZIONE

L'applicazione serverless sviluppata permette a un utilizzatore (ad esempio una rete di sensori) di inviare mediante richieste HTTP dei dati da far preprocessare, per poi essere inseriti in uno storage che verrà consultato da un'altra entità al fine di effettuare processamento vero e proprio dei dati.

La fase di preprocessing consiste nel validare, trasformare e memorizzare i dati - ovvero l'entità che consulterà la tabella finale si aspetta i dati in un certo formato e che siano presenti almeno un certo subset di dati che ci si aspetta in totale.

Pertanto la fase di *validazione* si occupa di verificare che i dati siano "validi", cioè, se ad esempio un certo dato deve rispettare un certo range, si verifica che quel dato rientri in quest'intervallo. Se i dati mancanti/errati non sono importanti, il processo di validazione può "decidere" di lasciar proseguire ugualmente la transazione.

La fase di *trasformazione* consiste appunto nel trasformare dati, ovvero, per esempio se si ha una temperatura, portarla da gradi celsius a kelvin, trasformare le stringhe da upper case a lower case, etc etc…

La fase di *store* memorizza in un certo storage (che può essere un DB NoSQL, data warehouse,…) la tupla di dati processata correttamente.

In caso contrario, per qualunque errore di preprocessing, va predisposta una soluzione sfruttando uno storage di appoggio per ogni fase che impedisca la perdita dei dati originali e permetta la segnalazione dello stato di una transazione: se fallisce la fase di transform, lo storage di appoggio di transform e validate vanno aggiornati (la fase store non viene mai raggiunta in questo esempio) in modo da poter essere consultati, localizzzare la causa dell'errore e non perdere i dati. Al contrario, se una transazione va a buon fine, tutti gli stati corrispondenti negli storage di appoggio indicheranno la buona riuscita dell'operazione.

2 BACKGROUND

Per realizzare l'applicazione sono stati utilizzati i servizi di AWS [1] consigliati (API gateway [5], Step Function [9], Lambda [7], DynamoDB [6]). In più stato utilizzato AWS Secrets Manager [8] per memorizzare in uno storage crittografico gestito da AWS, la chiave di autenticazione per le richieste di preprocessing di dati (se abilitato in fase di deployment). E' stato sfruttato l'AWS SDK Go v2 [3] per realizzare un'applicativo che automatizzi il deployment dell'infrastruttura completa. Per simulare un ipotetico sensore che manda dati da preprocessare, è stato realizzato un applicativo sempre in Go, che scarica da una mia cartella pubblica di Google Drive il dataset yellow taxis di New York [16] (già convertito da PARQUET a CSV), quindi lo memorizza nel computer che deve simulare le richieste di preprocessing: legge il CSV, "sporca" le entry automaticamente e effettua la richiesta HTTP verso l'endpoint. Dato che è quasi impossibile che falliscano le fasi di transform e store (deve fallire il servizio DynamoDB di AWS), si è deciso ai fini dell'osservazione del comportamento di SAGA serverless di implementare una soluzione che simuli fallimenti random tra le varie fasi. Non è stato utilizzato alcun build system in particolare ma solo shellscript e batch per Windows.

3 PROGETTAZIONE DELLA SOLUZIONE

Il fulcro dell'applicazione è ovviamente la state machine. Quest'ultima coordina le funzioni serverless al fine di compiere un determinato task. **La state machine corrisponde all'orchestrator del pattern SAGA**. Dopo aver triggerato l'esecuzione della state machine alla quale viene passata in input la tupla da preprocessare, sempre la state machine inoltra questa tupla, a sua volta, in input, alla prima funzione serverless che corrisponde alla fase di validate. Quando la funzione serverless di validate termina la sua esecuzione, la state machine ne valuta l'output (effettua una decisione) e decide se invocare la funzione serverless di transform (fase successiva del preprocessing) o la funzione serverless che permette di eseguire il "rollback". Ovviamente, tutto questo "a cascata", nel senso che se poi fallisce transform, dovrà essere invocata sia la funzione serverless che effettua "rollback" per la fase di transform che la funzione serverless che lo fa per la fase di validate. Nel path d'esecuzione della state machine, l'unico caso in cui c'è terminazione con successo è quando la funzione serverless store adempisce al suo compito, ovvero inserire la tupla nello storage finale. Sempre prendendo come riferimento il pattern SAGA "standard", **le singole funzioni serverless corrispondono ai microservizi** e **i "db di appoggio" alle singole funzioni serverless ai "db locali" per ogni microservizio** (ad esempio un payment db per un payment service, …).

La state machine **non processa eventi ma esiti delle funzioni serverless** e in base a ciò le orchestra. Rispetto alle situazioni normali di applicazione del pattern SAGA, ci troviamo in un caso diverso per dominio d' applicazione: SAGA si adatta bene quando abbiamo a che fare con molteplici microservizi (vale anche per le funzioni serverless) che "collaborano" per realizzare un applicazione - ad esempio in uno store si hanno i servizi "payment", "shipping", "billing" e "stock" con i relativi database accessibili solo ai rispettivi servizi: la transazione SAGA permette, al fine di realizzare il task di processare l'ordine di un utente, di effettuare una transazione distribuita che coordini i microservizi indipendenti, e non lasci i database con visibilità limitata ai microservizi inconsistenti (ad esempio un payment ricevuto ma lo stock ha lo stesso numero di articoli disponibili, quando in realtà, se il pagamento va a buon fine, lo stock dovrebbe avere un numero di articoli ridotto in unità). Se un'ordine viene effettuato, tutti i database di competenza di ogni servizio devono essere, nel loro dominio, aggiornati e coerenti. Event sourcing invece è un pattern che consiste in un db aggiuntivo (event store) per ogni microservizio e in pratica fa da log (append-only, compensativo) - prima di eseguire una certa azione, il microservizio scrive nell'event store. In questo modo, è possibile ricostruire lo stato del db dei pagamenti rieffettuando le operazioni listate nell'event store corrispondente. Nel nostro caso, la soluzione può essere snellita e semplificata: abbiamo a che fare con dei passaggi di preprocessing e quindi sarebbe inutile replicare più volte gli stessi dati per event sourcing (i "db di appoggio" fanno già da event store "semplificato"). Passando dalla state machine alle singole funzioni serverless, queste devono interagire con un loro db di appoggio per evitare la perdita delle tuple che ricevono in input e mantenerne uno stato del preprocessing della tupla che sia coerente con l'esito della transazione - se quest'ultima fallisce al livello i-esimo (sempre secondo struttura a cascata): dal livello i al livello 1 lo stato della tupla dovrà essere alterato indicando errore nel livello i-esimo (nella pratica l'alterazione avviene chiamando delle funzioni serverless che interagiscono con il medesimo database, ad esempio la funzione serverless `validate` e `flagValidateFailed` interagiranno con `validationStatus`). Se la transazione va a buon fine, nessuna azione deve essere intrapresa, la state machine termina, i dati preprocessati sono stati inseriti nel db finale e tutti i db di appoggio indicano per la corrispondente tupla lo stato di successo del preprocessing. Ovviamente, la funzione serverless, dopo aver inserito la tupla che ha ricevuto dalla precedente funzione della cascata, nel suo db di appoggio, deve portare a termine la fase di preprocessing di sua competenza e quindi restituirla in output (eventualmente modificata) per poi permettere alla successiva funzione serverless della cascata di effettuare le stesse operazioni, e cosi via…E' stato considerato molto importante, specificare nella progettazione, che le funzioni serverless devono prima di ogni altra cosa, inserire nel proprio db di appoggio, l'entry ricevuta in input dalla funzione al livello precedente della cascata, e poi effettuare il resto delle operazioni. Questo per garantire che i dati critici non vengano persi e quindi aumentare l'affidabilità del sistema. Le funzioni serverless aggiuntive per segnalare lo stato di failure nel corrispondente db di appoggio sono state necessarie per evitare duplicazione di codice dato che, se fallisce `validate` allora sarà necessario cambiare l'entry corrispondente in `validationStatus`, ma bisognerà fare la stessa identica cosa se fallisce `transform`, chiamando prima `flagTransformFailed`, poi `flagValidateFailed` (permettendo quindi di chiamare la stessa funzione invece che duplicare codice, ma questa è una questione implementativa). Potremmo quindi dire che il pattern è un misto tra SAGA e Event sourcing: i db di appoggio sono più che altro dei log che permettono di capire perchè una tupla di dati è errata (viene indicata la fase che ha causato fallimento, nello stato della transazione) e comunque non perderla perchè contenente dati critici. Il db di appoggio non è append-only e non compensativo (errori provocano la modifica dell'entry della transazione attuale, non l'aggiunta di un'ulteriore entry che ne revoca la riuscita del processo di preprocessing). Sarà poi necessario esporre un'API REST (HTTP) che dovrà, in qualche modo accettare come parametro la tupla e quindi permettere di avviare l'esecuzione della state machine. Considerando il deployment, è stato ritenuto utile dal punto di vista pratico realizzare un applicativo ad-hoc per creare l'infrastruttura su AWS, l'applicativo deve permettere di effettuare il setup dell'infrastruttura, il teardown, l'update delle funzioni lambda (utile in fase di sviluppo), e, su richiesta dell'utente, l'abilitazione dell'autenticazione per effettuare le richieste verso la REST API (evitando inserimenti non autorizzati e riducendo notevolmente, ma non azzerando i costi in caso di attacco di tipo denial of service, in quanto verrebbe eseguita continuamente la lambda di verifica autorizzazione e consultato allo stesso ritmo da quest'ultima il secrets manager per il recupero della chiave d'autenticazione).

4 DETTAGLI DELLA SOLUZIONE

Tenendo in mente la guida [12] di AWS, è stata implementata la state machine dall'editor grafico della console AWS e quindi ne è stata esportata in formato JSON la definizione [11] per supportare il deployment automatico. Scendendo nei dettagli: la state machine è in grado di effettuare branching sia a seconda del valore di ritorno della funzione lambda appena eseguita, sia se la funzione lambda incontra una situazione d'errore (per i linguaggi che le supportano, come Java o C++, parliamo di eccezioni, in Go si tratta di molto più semplici, ma comunque significativi, valori di tipo error [4] ritornati dalla lambda che la state machine è in grado di "leggere" e trattare): in alcuni casi può verificarsi un **errore** come la non possibilità di caricare l'SDK di AWS da parte della lambda o errori legati a DynamoDB. Nel passo specifico di una lambda si può indicare alla state machine come gestire errori/eccezioni inaspettati. Proseguendo invece con il regolare execution flow della state machine, al passo successivo, si può inserire un blocco decisionale che confronta un predicato su chiavi di un JSON object **restituito** dalla lambda appena eseguita. E' stato poi utilizzato il costrutto parallel per le funzioni lambda che devono indicare le entry della transazione fallita: non c'è motivo di attendere la terminazione della lambda che segnala l'entry nel db di appoggio di `transform` come fallita, per eseguire poi la lambda che segnala l'entry nel db di appoggio di `validate` come fallita - possono essere eseguite in parallelo, diminuendo cosi il tempo d'esecuzione della state machine.

*Sottolineamo quindi che un **fallimento** è causato da **errore** (assimilabile a eccezione in linguaggi che le supportano, invece nel nostro caso, ovvero Go, è la terminazione della funzione lambda con* `err != nil`*) o **valore di ritorno** inaspettato (ma* `err == nil`*)*

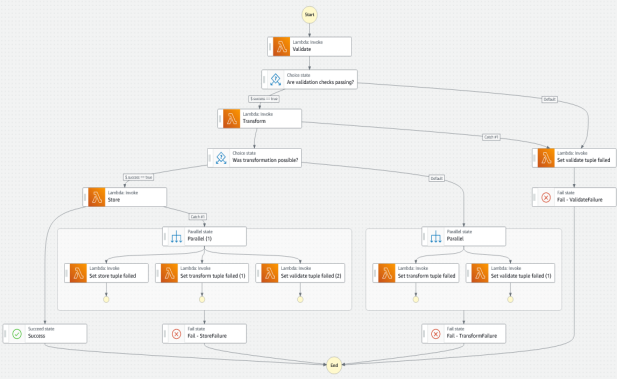

Figura 1: Definizione state machine

Scendendo nei dettagli della gestione dei **fallimenti**:

- per `validate`: se va incontro ad un **errore** (`err != nil`) allora non viene eseguita la `flagValidateFailed` perchè in ogni caso non è stato possibile inserire la tupla nel suo db di supporto e quindi non c'è nulla da cambiare (situazione molto improbabile, fallimenti legati a DynamoDB e/o al setup della config. di AWS). Se invece il **valore di ritorno** non è quello che ci si aspetta (`"success": true`) allora si, la state machine esegue `flagValidateFailed` perchè il preprocessing è fallito (molto probabile).
- per `transform`: se va incontro ad un **errore** (`err != nil`) allora non viene eseguita la `flagTransformFailed` ma solamente la `flagValidateFailed`, perchè se la state machine arriva alla lambda `transform` significa che la precedente `validate` è terminata con successo (c'è una tupla nel db di appoggio di `validate` con stato attuale di successo che va cambiato) mentre, dato che la lambda `transform` è terminata con **errore** (`err != nil`) questo implica per forza che l'inserimento nel db di appoggio di `transform` non è avvenuto e quindi non c'è nulla da modificare in quest' ultimo (situazione molto improbabile, sempre fallimenti legati a DynamoDB e/o al setup della config. di AWS). Se invece il **valore di ritorno** non è quello che ci si aspetta (`"success": true`) allora si, la state machine esegue sia `flagValidateFailed` che `flagTransformFailed` (c'è stato inserimento nel db di appoggio di `transform`) in parallelo, perchè il preprocessing è fallito (situazione particolare, difficile che transform fallisca perchè non riesce a effettuare una trasformazione).
- per `store`: se va incontro ad un **errore** (`err != nil`) allora viene eseguita la `flagStoreFailed`, `flagTransformFailed` e `flagValidateFailed` in parallelo. Una particolarità di questa lambda è che oltre al suo db di appoggio nel quale registra la tupla processata dalla precedente `transform`, deve poi creare un nuovo item nella tabella DynamoDB finale vera e propria. La tupla passata in input a `store` è rappresentata mediante una stringa che è una entry di un file CSV con carattere di separazione la tabulazione: per ognuno di questi ultimi vengono creati due attributi nell'item di DynamoDB. In questo caso la situazione di **errore** (`err != nil`) può rappresentare due situazioni:
  - l'inserimento della tupla ricevuta nel **db di appoggio** è fallito (di conseguenza non è stato possibile inserire nemmeno l'item nel db finale): tupla nel db di appoggio non presente
  - l'inserimento dell'item nel **db finale** è fallito (ma è stato possibile inserire la tupla nel db di appoggio): tupla nel db di appoggio presente

  Ricordando che entrambe le situazioni sono estremamente improbabili dal verificarsi (come al solito, errori legati a DynamoDB di Amazon), è stato ritenuto inutile aggiungere ulteriore specifica di rilevamento d'errore alla state machine perchè richiederebbe ulteriori tipi di errore [4] per discriminare le due situazioni sopra descritte, e intraprendere un path d'esecuzione piuttosto che un altro: si introduce troppa complessità per un problema che si verifica solo se fallisce l'infrastruttura di AWS e questo è estremamente improbabile, oltretutto, se anche venisse eseguita `flagStoreFailed` e non c'è l'entry da aggiornare allora non cambia nulla - l'errore generato dalla libreria di DynamoDB all'interno della lambda di flagging viene ignorato dalla state machine. L'unico svantaggio è comunque l'esecuzione di una lambda "a vuoto", ma come già scritto sopra, è estremamente improbabile che ciò accada, e se anche accadesse i costi sono irrisori e comunque non ci sarebbero conseguenze per le tabelle DynamoDB (non viene creata una entry se non è già presente). La lambda `store`, dato che può fallire l'inserimento solamente a causa di errori di DynamoDB, può avere come **valore di ritorno** solamente `"success": true` (oppure errori come descritto sopra).

L'API gateway è configurato in modo da ricevere una richiesta HTTP `"POST /store"` e:

- **se è abilitata l'autenticazione**: invocare la lambda `authorizer` che verifica le credenziali - l'API Gateway passa informazioni alla lambda - in Go c'è un tipo specifico offerto dalle librerie di AWS Go SDK v2 che contiene il campo `IdentitySource`, quindi viene confrontata la chiave fornita in richiesta HTTP (in una entry con chiave `"Authorization"` e valore la auth key immessa dall'utente, dell'header HTTP) con quella immagazinata nell'archivio crittografato gestito da AWS secrets manager (impostata in fase di deployment). Se il controllo ha successo, prosegue, altrimenti mostra un errore della classe 400 HTTP
- **se non è abilitata l'autenticazione** - prosegue

In particolare in fase di deployment devono essere unite route (`POST /store`) e integration con servizio interno di Amazon AWS (Step Function, identificato dall'ARN) per triggerarne un certo task (`StartExecution` [10]) dopo l'API gateway riceve la richiesta `POST /store`. Il payload della richiesta HTTP è in formato JSON e viene forwardato alla state machine (è praticamente la tupla in input). Invocata poi a sua volta, dalla state machine, e passata alla lambda `validate`, la tupla in formato raw (CSV entry), viene immediatamente ricavato dal clock di sistema l'istante di invocazione dell'handler della lambda, quindi più tardi viene usato per

ricavare un ID univoco che serve a rappresentare la transazione (per permettere di effettuare rollback in caso di errore, ovvero effettuare flagging della transazione fallita in fase attuale o successive) - viene scambiato tra le varie lambda ed è ovviamente lo stesso in tutti i db di appoggio (`validationStatus`, `transformationStatus`, `storeStatus`).

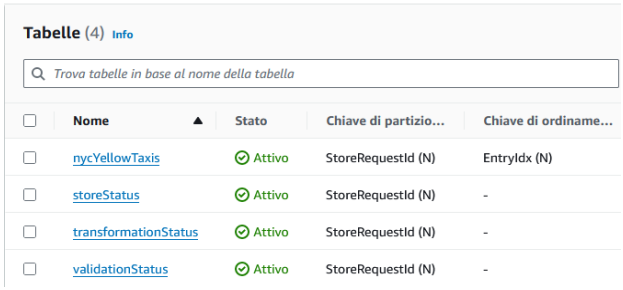

Figura 2: Tabelle DynamoDB di appoggio e la tabella finale

E' stato necessario perchè:

- usare identificativi/indici del dataset non è affidabile e potrebbe essere inviata più volte in istanti diversi la stessa tupla
- usare un identificativo incrementale non è affidabile: supponendo che la `validate`, per assegnare ID, debba (in modo poco efficiente) prelevare l'ID più alto presente attualmente nella tabella (e quindi incrementarlo per l'item che sta per essere inserito), allora se arrivano due richieste in contemporanea (e quindi due lambda `validate` il cui flusso va di pari passo) entrambe le query verso la tabella DynamoDB restituiscono per entrambe le lambda, lo stesso identico insieme di items → errori nell'inserimento e perdita di dati perchè per entrambe le transazioni (diverse) si ottiene lo stesso ID (stesso max{IDs of items} + 1).

L'ID univoco della transazione viene quindi ricavato mettendo insieme l'istante esatto di invocazione dell'handler registrato della lambda `validate`, il contenuto stesso della tupla passata in input (è un funzione hash estremamente rapida) e un numero intero random da 0 a 10000 il problema della collisione degli ID è tuttavia sempre possibile ma deve esserci una condizione sfortunata secondo la quale l'epoch unix ricavato è lo stesso per entrambe le transazioni (che avvengono in contemporanea), la funzione hash collide per due tuple diverse e il numero intero estratto dal PRNG è uguale a quello dell'altra trasazione. Una possibile alternativa (che riduce fortemente la possibilità di collisione anche se le transazioni avvengono in contemporanea), è ottenere l'epoch unix non in secondi ma in nanosecondi, tuttavia, dato che quest'ultimo viene sommato all'output della funzione hash calcolata sulla tupla, e al numero random si vuole evitare ogni possibilità di integer overflow (per evitare undefined behaviour, ...viene ospitato l'ID univoco di transazione in un intero unsigned a 64 bit, anche se l'identificativo ne richiede più o meno la metà).

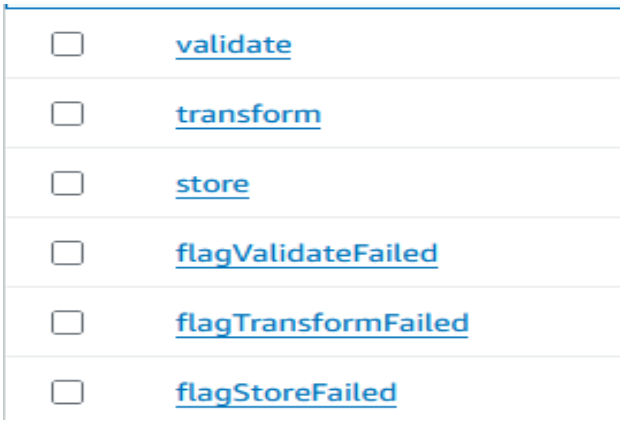

Figura 3: Lambdas

Dopo aver inserito la tupla nel db di appoggio di `validate` (tabella DynamoDB `validationStatus`), con stato di transazione successo (ovvero, 0) e il suo ID univoco appena calcolato, vengono effettuati i check sulle colonne del dataset al fine di validarne la correttezza, e questo viene fatto secondo la documentazione di NYC [15] sul significato delle colonne del dataset. Si sfrutta la registrazione di callback: quest'ultima ritorna `true` o `false` a seconda se il check sulla colonna passatagli in input va a buon fine o se fallisce (e quindi la tupla non è valida) - in alcuni casi, se i dati non sono ritenuti fondamentali, la callback può comunque ritornare true e lasciar passare (annullando il campo, stringa vuota) la tupla alla prossima fase di preprocessing. Ulteriori check di validazione sulla tupla sono:

- Se la tupla in input alla lambda `validate` è vuota, quest'ultima termina immediatamente, col valore di ritorno che segnala fallimento (il check è posto dopo l'ottenimento del valore del clock di sistema attuale)
- Se il numero di colonne della tupla in input è diversa da quella che ci si aspetta, la lambda `validate` termina (valore di ritorno segnala fallimento)
- Vengono effettuati anche controlli "cross-columns" per controllare constraint sulle colonne (es. se una colonna è la somma di altre).

Se la validazione ha successo, viene ritornata dalla lambda `validate` la tupla validata (eventualmente con uno o più campi vuoti, se non validi ma ritenuti non importanti) in un JSON object (insieme a `"success": true` e l'ID univoco della transazione). La lambda `transform` è responsabile, nella catena di preprocessing, di trasformare i dati da una certa rappresentazione a un'altra equivalente. Riceve dalla state machine, la tupla validata precedentemente con successo da `validate`: quindi, dopo aver immagazinato nel suo db di appoggio (`transformationStatus`) la tupla con l'ID della transazione (passato in input dalla state machine: lo stesso generato da `validate`) e il suo stato (posto inizialmente a 0, ovvero successo), inizia a effettuare trasformazione dei dati:

- Cambia carattere di separazione della tupla CSV da ',' a '\t'
- Valori degli attributi da numerici a stringa secondo il data dictionary: [15]
- Trasformazione di data e ora in formato diverso

- Valori float del tipo '10.0', '123.0', ... trasformati direttamente in interi

Problema della transform è la difficoltà nel causare naturalmente un fallimento, ai fini del testing è un problema che viene affrontato in sezione successiva. In caso di trasformazione con esito positivo, viene ritornato il JSON object cosi come descritto per `validate`. La lambda `store`, infine, riceve in input la tupla trasformata e l'ID della transazione, quindi immediatamente la inserisce nel proprio db di appoggio (`storeStatus`): la tupla viene splittata secondo il carattere di tabulazione e viene creato un nuovo item nella tabella DynamoDB finale di consultazione (`nycYellowTaxis`) che però abbia tutti gli attributi separati. Non si ha più la necessità di memorizzare lo stato della transazione, tuttavia ne mantenuto l'ID univoco, per differenziare entry diverse ma con valori uguali (per esempio se fossero dati di una rete di sensori ambientali: dispositivi registrano la stessa CO_2 nell'aria in istanti diversi). Anche in questo caso, risulta difficile sperimentare gli effetti sulla transazione SAGA in caso di fallimenti su questo livello. La lambda `authorizer` invece è "aggiuntiva" ed opzionale: è già stata descritta in precedenza e utilizza un ulteriore servizio di AWS che è un archivio crittografico "AWS Secrets Manager" [8] L'injector è un tool sviluppato in Go che consente automaticamente di iniettare dati leggendo da un file CSV (nel nostro caso il file è https://d37ci6vzurychx.cloudfront.net/tripdata/yellow_tripdata_2024-02.parquet, che è stato convertito in CSV e quest'ultimo caricato su Google Drive) che viene automaticamente scaricato dal web e memorizzato nel fs (in modo da non doverlo scaricare più volte). Questo tool toglie allo sviluppatore l'onere di sporcare il file CSV a mano (con la possibilità comunque di evitare di sporcare i dati specificandolo da command line), facendolo in place (il file CSV non viene toccato, vengono sporcate le entry caricate in memoria) e in maniera randomica (un PRNG uniforme e un valore di soglia permettono di decidere se sporcare o meno), secondo regole personalizzabili (struct e callback) sia per-colonna (cambio valori a caso, ad esempio da $1 \longrightarrow 2$ o da "username" $\longrightarrow$ 3.14), sia "tuple-wise" (ad esempio aggiungendo virgole a caso, cambiando caratteri di separazione, ...), è possibile regolare il rate di immissione delle richieste verso l'API endpoint per ovviare a problemi legati ai costi e/o limitare il grado di concorrenza delle lambda. Sfruttando la direttiva replace dei moduli Go [14] vengono realizzate le lambda che si basano sullo stesso modulo: `flagPhaseFailed`: `flagValidateFailed`, `flagTrasformFailed` e `flagStoreFailed` sono molto simili quindi non ha senso duplicare codice - vengono implementate chiamando una funzione esportata dalla libreria `flagPhaseFailed` - tutte e 3 le lambda condividono lo stesso codice.

5 TESTING

Per testare la soluzione, è necessario effettuare il deployment su AWS con il tool apposito sviluppato in Go e fornito insieme al resto del codice: quindi prelevare dalla dashboard di AWS instructure le credenziali/security token della sessione AWS avviata e salvarle nel proprio PC. Testare questo sistema consiste nell'eseguire richieste HTTP con payload una tupla in input che viene quindi preprocessata e quindi osservare i dati nelle tabelle DynamoDB, il punto è che però la transazione può fallire su una certa fase e quindi ci si aspettano determinati cambi di stato nelle tabelle DynamoDB alle entry corrispondenti alla stessa transazione - questo comportamento è facilmente osservabile solamente sulla `validate` - è quasi impossibile far fallire la `transform` o la `store` (come già scritto, `transform` può fallire se per qualche motivo la trasformazione di dati non è possibile - che nel nostro caso è impossibile dato che si tratta di semplici trasformazioni - o se fallisce DynamoDB, per `store` è ancora peggio perchè può fallire solo se fallisce DynamoDB). Al fine di osservare il comportamento della transazione SAGA serverless, si utilizzano le Go build constraints [13] per poter passare al compilatore Go una flag che permette di definire un valore che permette di decidere se includere nel processo di compilazione una certa unità di traduzione (pre-compile time). Nel nostro caso, usando anche la direttiva replace dei moduli Go [14], è stato possibile realizzare un modulo "failsim" che:

- SE **VIENE** DEFINITO `ENABLE_FAILSIM` tramite il compilatore Go: viene generato del codice che chiama un PRNG e a seconda del valore ritornato ritorna un error non-nil oppure error che sia nil
- SE **NON VIENE** DEFINITO `ENABLE_FAILSIM` tramite il compilatore Go: viene generato del codice che ritorna esclusivamente errror nil

Lo svantaggio di questo approccio è l'inclusione, in ogni caso, di codice aggiuntivo che incrementa la grandezza delle lambda e un leggero overhead dovuto alla chiamata a funzione **OopsFailed()**, check del valore di ritorno di quest'ultima (a prescindere dall'abilitazione della funzionalità) e la chiamata al PRNG più check valore threshold (se la funzionalità viene abilitata). La chiamata a "failsim" è stata posizionata (nelle lambda `validate`, `transform` e `store`) prima di richieste verso DynamoDB (simulando fallimenti in DynamoDB), dopo check di successo delle trasformazioni, ma non nel check di validità della tupla in quanto già failure-prone. Ovviamente, il concetto è che se `failsim.OopsFailed`, posizionato in determinati punti, ritorna un errore diverso da `nil` allora causa **fallimento** per **errore** o **valore di ritorno** inaspettato.

6 RISULTATI

Per questo progetto, per risultati, si intendono gli esiti delle transazioni, lo stato delle entry dei db di appoggio e l'osservazione delle entry del db finale, consultabile da un ipotetico applicativo client/consumer/... che effettuerà il processamento vero e proprio dei dati. Il risultato finale è positivo se per tutti i db di appoggio, tutte le entry, ognuna corrispondente a una certa transazione eseguita, corrispondono allo stato che ci si aspetta in base alla tupla inviata alla pipeline dati e i db di appoggio devono essere coerenti: ad esempio se c'è fallimento in `transform` allora:

- deve essere presente la entry con la tupla passata in input, con ID di transazione sia nel db di appoggio di `validate` che in quello di `transform` (anche se la presenza o meno nel db di appoggio di `transform` dipende da quando avviene il fallimento), ma non in `store` (fase della pipeline mai raggiunta)
- la tupla deve avere codice di errore corrispondente a fallimento in fase di `transform` in entrambi i db di appoggio
- nel db finale non deve esistere alcuna corrispondenza di questa tupla che non ha passato tutte le fasi di preprocessing

Per collezionare le informazioni si utilizza la console grafica webapp di AWS per DynamoDB e le Step Functions (oss. si ha un costo per

effettuare query su tabelle DynamoDB)

Se considerassimo una state machine la cui esecuzione termina con successo:

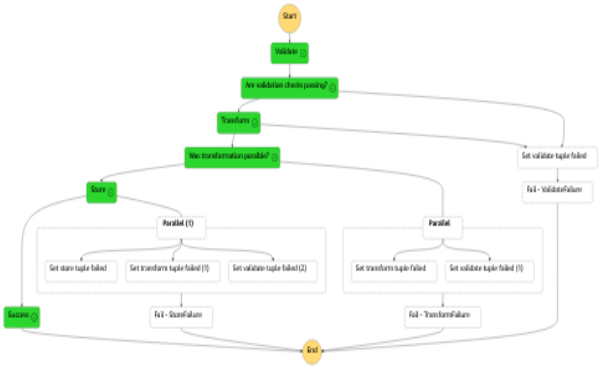

Figura 4: Esecuzione di una state machine terminata con successo

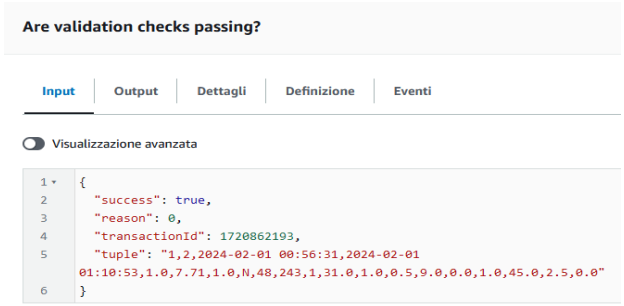

Figura 5: ID della transazione in input al blocco decisionale dalla lambda `validate`

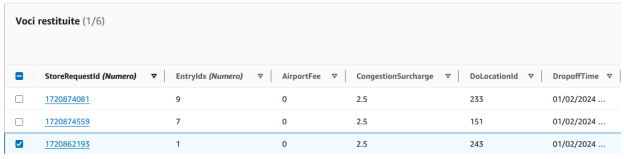

Figura 6: Item aggiunto alla tabella DynamoDB finale (`nycYellowTaxis`)

Invece, ad esempio, una state machine che termina con fallimento in `transform`:

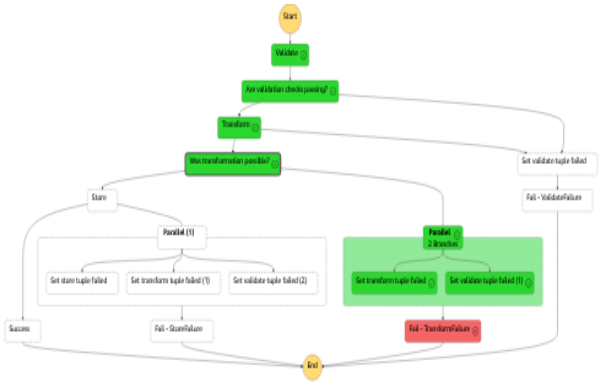

Figura 7: Esecuzione di una state machine terminata con fallimento in `transform`

Allora nei db di appoggio `validationStatus` e `transformationStatus`, la situazione verrebbe indicata:

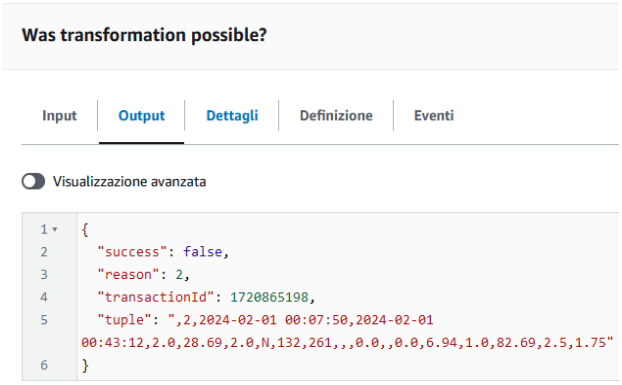

Figura 8: ID della transazione in input al blocco decisionale dalla lambda `transform`

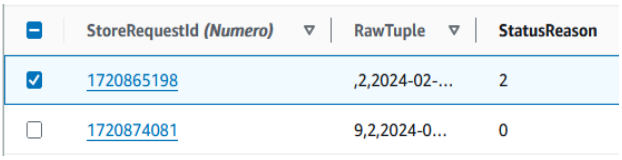

Figura 9: Entry che indica `StatusReason = 2` per questa transazione nella tabella di appoggio `transformationStatus`

Figura 10: Entry che indica `StatusReason = 2` per questa transazione nella tabella di appoggio `validationStatus`

Altri due esempi importanti sono:

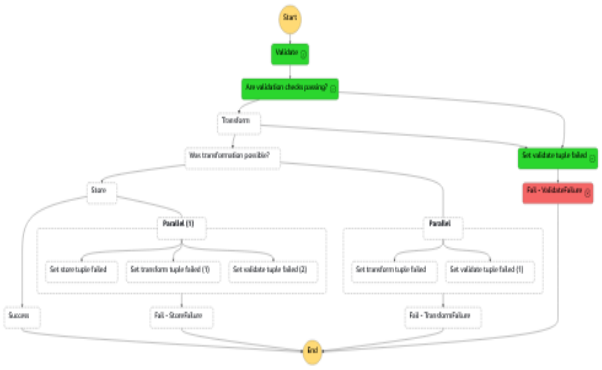

Figura 11: Esecuzione di una state machine terminata con fallimento in `validate`

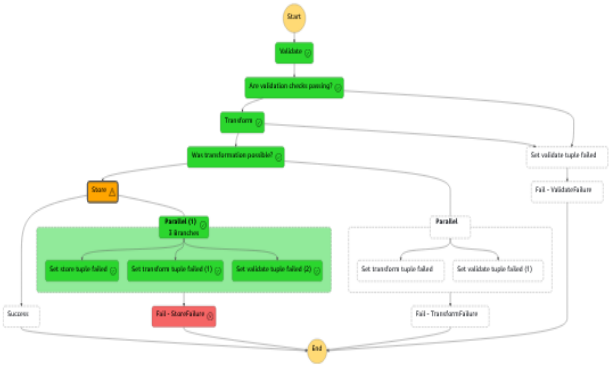

Figura 12: Esecuzione di una state machine terminata con fallimento in `store`

Ovviamente, altre casistiche sono possibili (ad esempio in transform un fallimento causato da un errore impedisce l'inserimento nella sua tabella DynamoDB di appoggio).

L'attributo `StatusReason` nelle tabelle DynamoDB di appoggio può assumere i seguenti valori:

- 0 se la transazione è terminata con successo
- 1 se la transazione è fallita nella fase `validate` (solo in `validationStatus`)
- 2 se la transazione è fallita nella fase `transform` (solo in `transformationStatus` e `validationStatus`)
- 3 se la transazione è fallita nella fase `store`

Ovviamente, nella tabella DynamoDB finale vanno a finire solamente le tuple preprocessate correttamente, e in tutti i db di appoggio delle fasi di preprocessamento affrontate, lo stato indicherà terminazione della transazione con successo (transazione corrispondente individuabile con `StoreRequestId`).

Se al contrario una tupla non viene preprocessa correttamente, allora, la tupla sarà presente al più (se per esempio la `transform` non dovesse riuscire a inserire nel proprio db di appoggio la tupla ricevuta in input, errori legati a DynamoDB) nel db di appoggio della fase di preprocessamento in cui è fallita e nei db di appoggio delle fasi precedenti ad essa (a ritroso, fino alla prima) indicando in TUTTI i db di appoggio lo stato di fallimento in quella fase per la transazione fallita (se ad esempio fallisce la `transform`, allora sia il db di appoggio di `transform`, che il db di appoggio di `validate` indicheranno terminazione con errore al livello 2 per la transazione fallita)

7 DISCUSSIONE

Non sono stati riscontrati particolari problemi nella realizzazione dell'applicativo: sicuramente, al fine di testare il funzionamento dell'applicativo serverless e dell'implementazione del pattern SAGA, è stato speso del tempo per pensare a come far fallire una fase di trasformazione dei dati che difficilmente fallisce se il dato "di partenza" è stato validato - quindi se ne è venuti a capo con la soluzione "failsim". Però failsim stessa ha dei problemi: il PRNG di Go genera numeri random con distribuzione uniforme, quindi di fatto avere una "threshold" non ha molto senso (tutti i numeri interi da 0 a N sono equiprobabili) - nonostante ciò, si hanno due opzioni:

- quando si confronta il risultato del PRNG con una threshold, al fine di determinare se causare errore o meno, usare = e quindi diminuire notevolmente il numero di fallimenti simulati, richiedendo però un maggior numero di runs della state machine per generare tutti i casi possibili (es. `if PRNG_IntN(0,100) == 60 then return error; else return success;`)
- al contrario, quando si confronta il risultato del PRNG con una threshold, al fine di determinare se causare errore o meno, usare $\leq$ o $\geq$ e quindi aumentare il numero di fallimenti simulati, richiedendo un minor numero di runs della state machine per generare tutti i casi possibili (es. `if PRNG_IntN(0,100) >= 60 then return error; else return success;`)

E' stata scelta la prima opzione, e questo ha richiesto l'immissione di un numero di richieste notevoli al fine di generare i casi di fallimenti più importanti. E' stato oltretutto necessario limitare il rate di immissione delle richieste HTTP al fine di evitare la disattivazione automatica dell'account AWS da parte di instructure. In futuro, si può sicuramente introdurre, una segnalazione degli errori più precisa nelle entry: in particolare si possono indicare nell'entry del db di appoggio la ragione del fallimento (es. "data e ora di formato sbagliato in componente 3 della tupla passata in input") e si può anche migliorare la gestione degli errori da parte della state machine per evitare chiamate inutili alle lambda.

8 LIBRERIE ESTERNE

Oltre alla libreria standard di Go:

- l'injector non ha dipendenze esterne
- tutte le lambda utilizzano:
  - "github.com /aws/aws-lambda-go/lambda" [2]
  - "github.com /aws/aws-sdk-go-v2/config" [3]
- SOLO la lambda authorizer, in più:
  - "github.com /aws/aws-lambda-go/events" [2]

  - "github.com /aws/aws-sdk-go-v2/service/secretsmanager" [3]
  - (type definitions) "github.com /aws/aws-sdk-go-v2/service/secretsmanager/types" [3]
- le lambda che compongono la state machine (validate, transform, store, flagValidateFailed, flagTransformFailed, flagStoreFailed), in più:
  - "github.com /aws/aws-sdk-go-v2/service/dynamodb" [3]
  - "github.com /aws/aws-sdk-go-v2/feature/dynamodb /attributevalue" [3]
- SOLO le lambda di flagging di errore (flagValidateFailed, flagTransformFailed, flagStoreFailed), oltretutto:
  - "github.com /aws/aws-sdk-go-v2/feature/dynamodb/expression" [3]
  - (type definitions) "github.com /aws/aws-sdk-go-v2/service/dynamodb/types" [3]
- il programma di deployment dell'infrastruttura AWS:
  - "github.com /aws/aws-sdk-go-v2/aws" [3]
  - "github.com /aws/aws-sdk-go-v2/config" [3]
  - "github.com /aws/aws-sdk-go-v2/service/apigatewayv2" [3]
  - (type definitions) "github.com /aws/aws-sdk-go-v2/service/apigatewayv2/types" [3]
  - "github.com /aws/aws-sdk-go-v2/service/dynamodb" [3]
  - (type definitions) "github.com /aws/aws-sdk-go-v2/service/dynamodb/types" [3]
  - "github.com /aws/aws-sdk-go-v2/service/iam" [3]
  - "github.com /aws/aws-sdk-go-v2/service/lambda" [3]
  - (type definitions) "github.com /aws/aws-sdk-go-v2/service/lambda/types" [3]
  - "github.com /aws/aws-sdk-go-v2/service/secretsmanager" [3]
  - (type definitions) "github.com /aws/aws-sdk-go-v2/service/secretsmanager/types" [3]
  - "github.com /aws/aws-sdk-go-v2/service/sfn" [3]

RIFERIMENTI BIBLIOGRAFICI

[1] Amazon AWS. https://aws.amazon.com/it/.
[2] AWS lambda (Go). https://pkg.go.dev/github.com/aws/aws-lambda-go.
[3] AWS SDK for Go v2. https://pkg.go.dev/github.com/aws/aws-sdk-go-v2.
[4] Go by Example: Errors. https://gobyexample.com/errors.
[5] Amazon AWS. API Gateway. https://aws.amazon.com/it/api-gateway/.
[6] Amazon AWS. DynamoDB. https://aws.amazon.com/it/dynamodb/.
[7] Amazon AWS. Lambda. https://aws.amazon.com/it/lambda/.
[8] Amazon AWS. Secrets Manager. https://aws.amazon.com/it/secrets-manager/.
[9] Amazon AWS. Step Functions. https://aws.amazon.com/it/step-functions/.
[10] Amazon AWS. Step Functions - Actions: StartExecution. https://docs.aws.amazon.com/step-functions/latest/apireference/API_StartExecution.html.
[11] AWS. Amazon States Language - AWS Step Functions. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html.
[12] Tabby Ward (AWS), Rohan Mehta (AWS), and Rimpy Tewani (AWS). Implement the serverless saga pattern by using AWS Step Functions. https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/implement-the-serverless-saga-pattern-by-using-aws-step-functions.html.
[13] Golang. build package - go/build - Go Packages. https://pkg.go.dev/go/build#hdr-Build_Constraints.
[14] Golang. Go Modules Reference - The Go Programming Language. https://go.dev/ref/mod#go-mod-file-replace.
[15] NYC. Data Dictionary – Yellow Taxi Trip Records. https://www.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf.
[16] NYC. TLC Trip Record Data. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page.