##### **Data Model**

The user, user history and attraction tables are in DynamoDB. We do not store restaurant data because they change often. The user table stores user preferences:

(username: str,

 favAttractionTypes: []str,

 favCountries: []str)

The user history table stores browsing history:

(username: str,

 attractionId: str,

 lastVisitTimestamp: int,

 frequency: int)

The attraction table stores attraction details:

(attractionId: str,

 address: str,

 attractionName: str,

 attractionType: []str,

 description: str

 opening\_hours: []str,

 photo\_url: str,

 rating: double,

 reviews: []str

 reviews\_cnt: int)

We also use ElasticSearch for convenient queries, with the following schema:

(attractionId: str,

 address: str,

 attractionName: str,

 attractionType: []str,

 description: str,

 visSimilarAttractionIds: []str,

 descSimilarAttractionIds: []str,

 rekognitionLabels: []str)

##### **Recommendation Architecture**

Since the recommendation process can be time-consuming, we follow a pre-computed approach. We first create a CloudWatch event which periodically triggers the recommender lambda function to update the recommendation results for all users. Once the results are ready, they will be written to a Redis cluster in ElastiCache which can be scaled conveniently. We generate and cache sufficient recommendations for each user (i.e., the user will not be able to finish viewing all of them in a short period of time). When "/feed" is requested, the recommender lambda function will first try to randomly sample 20-30 attractions from the cached results. The time-to-live (TTL) of cache is set to 0.5-2 minutes and is less than the CloudWatch trigger period. These parameters are tunable, depending on the tradeoff between read time and the freshness of recommendation results (i.e., reflecting the user’s recent browsing history). When a cache miss happens, a real-time recommendation process will be run, and once complete the results will also be cached.

Implementation-wise, the architecture is slightly more complicated than the architectural diagram. ElastiCache must be run inside a VPC, while the recommender lambda function needs access to DynamoDB and ElasticSearch which are not in a VPC. To avoid using the costly NAT gateways and to simplify future maintenance, an auxiliary lambda function that handles Redis get, set and delete requests, is created inside the Redis cluster’s VPC. An extra route "/cache" is created in API Gateway for the cache lambda function, allowing lambda functions outside the VPC to visit the cache.

Because our attraction data are constant once downloaded from external APIs, for simplicity, we perform some analysis offline once only: for each attraction, we obtain extra labels and the lists of similar attractions in terms of vision and language respectively, and write them to ElasticSearch when creating the index. More details can be found in the algorithm section. In the real world, if the data are not constant, the feature vectors can be written to ElasticSearch and online similarity analysis will be performed there instead.

##### **Recommendation Algorithm**

Our algorithm is a combination of rule-based recommendation and item-based collaborative filtering. The rule-based recommendation is solely based on the user's preferences on attraction types and countries. It is equivalent to the following SQL query:

SELECT \* FROM Attractions WHERE (prefType1 in attractionTypes OR prefType2 in attractionTypes OR …) AND (prefCountry1 in address OR prefCountry2 in address OR…)

For a newly registered user, if there is no preference at all, the system will randomly return attractions. If the user has not visited any attraction detail pages, the recommendation is purely rule-based. Rule-based recommendation is important when there is little knowledge about the user (namely the cold start problem).

Item-based collaborative filtering is involved after a user has viewed the detail page of some attractions. The basic idea is that if a user is interested in one place, then he/she might be interested in similar places as well. The query to get the recently viewed attractions is equivalent to the following:

SELECT attractionId FROM UserHistory

WHERE username=<username>

ORDER BY lastVisitTimestamp DESC, frequency DESC

LIMIT <limit>

We harness the types, picture, and description of an attraction to compute the similarity to another attraction. Getting attractions of the same types is straightforward. We take the union of all types of recently viewed attractions and use the most frequent 10 types to get a list of attractions.

Processing picture and description, on the other hand, requires some computer vision and natural language processing techniques. As for the picture, we compute two features: 1. use Rekognition to perform object detection and output text labels which will be used in the same manner as the attraction types mentioned above; and 2. pre-train convolutional neural networks such as ResNet50 and DenseNet161 on the Places 365 dataset with 10-million scene images, and for each picture concatenate the feature vectors produced by these models (equivalent to an ensemble) to form one feature vector to compute cosine similarity. The pre-training top-5 accuracies for ResNet50 and DenseNet are 85.9% and 86.5% respectively. Regarding description, we use a pre-trained BERT model to obtain the vector representation which will also be used in cosine similarity computation.

For each attraction, we then rank other attractions based on the cosine similarity values and pick the top-10. We want the recommendation results to reflect the most recent user activity. Thus, when merging multiple lists of similar attractions, we order the attractions similar to the more recently viewed attractions in front of those similar to attractions of earlier history.

To combine results from types, labels, visual features and language features, we apply some set operations. We first take the union of top similar attractions with respect to vision and language features. The reason why we take union rather than intersection is because we want the recommendations to be sufficient and diverse. Nonetheless, this may not be precise. For example, similar pictures may not indicate similar attraction types, especially for buildings. Moreover, some of the attraction pictures are noisy (e.g., a bicycle occupies many pixels on a park picture). Therefore, we refine this set by taking the intersection with attractions of relevant types and Rekognition labels.

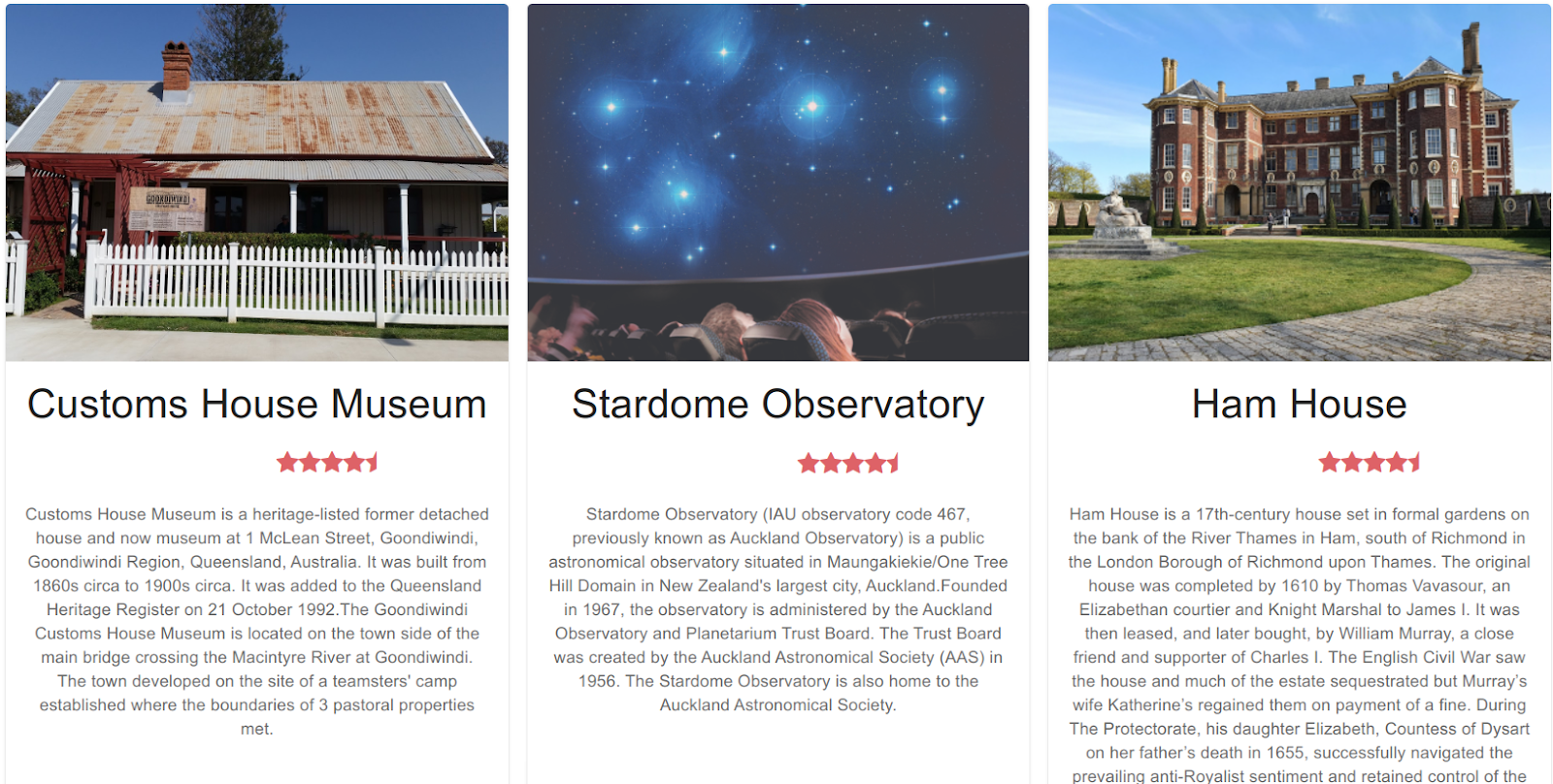
##### **Recommendation Ablation**

The collaborative filtering part of our recommendation algorithm may seem complicated, and thus we show some comparisons with different methods used in the recommendation process. Assume the user just viewed the detail page of a car museum located in New Zealand shown below.



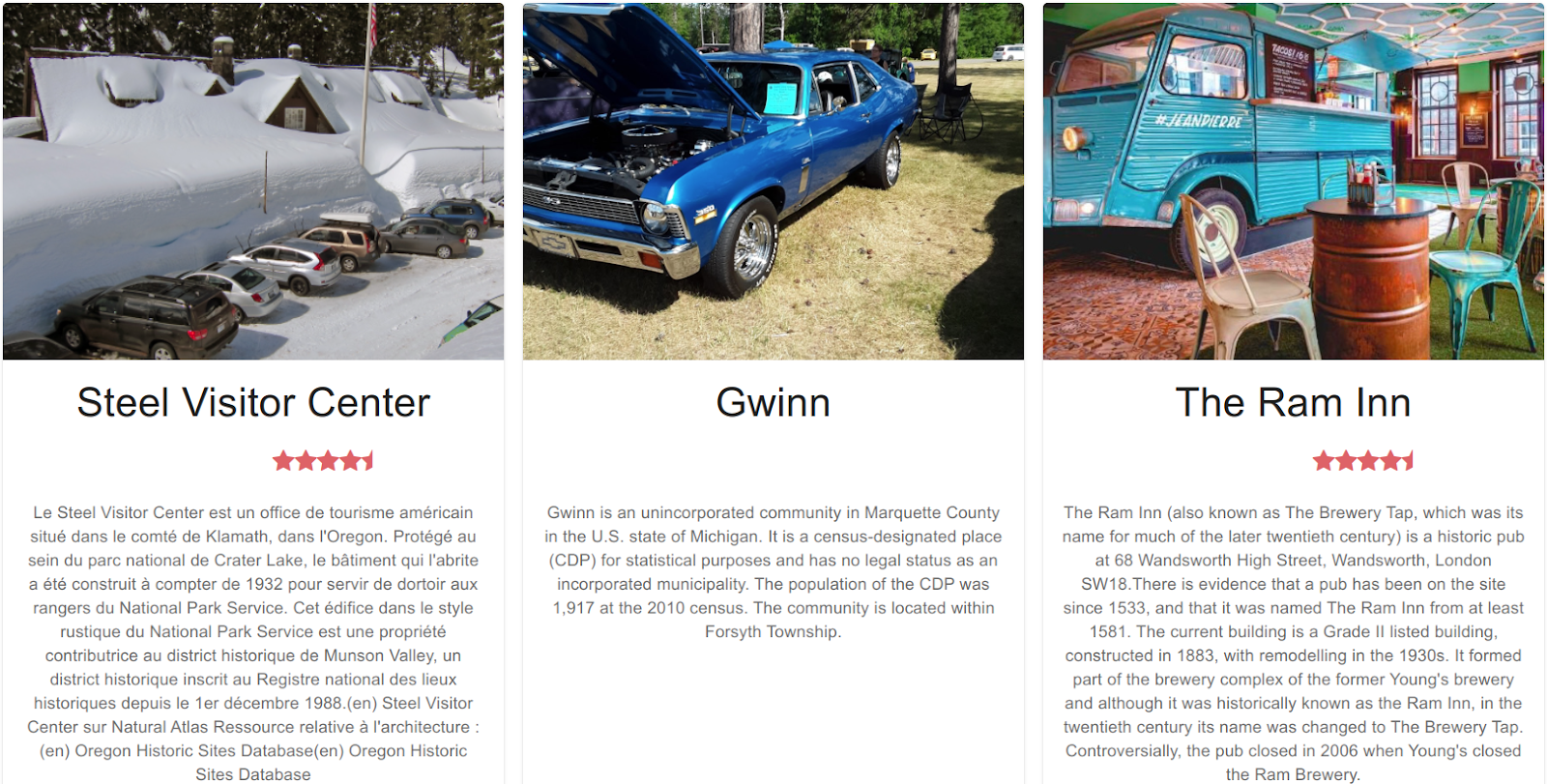
The following figures show some returned results from different configurations of item-based collaborative filtering. We define the type-related method, the visual feature only method and the language feature only method as the baselines, and compare them with visual+language feature models with/without refinement.

*Relevant Types Only*



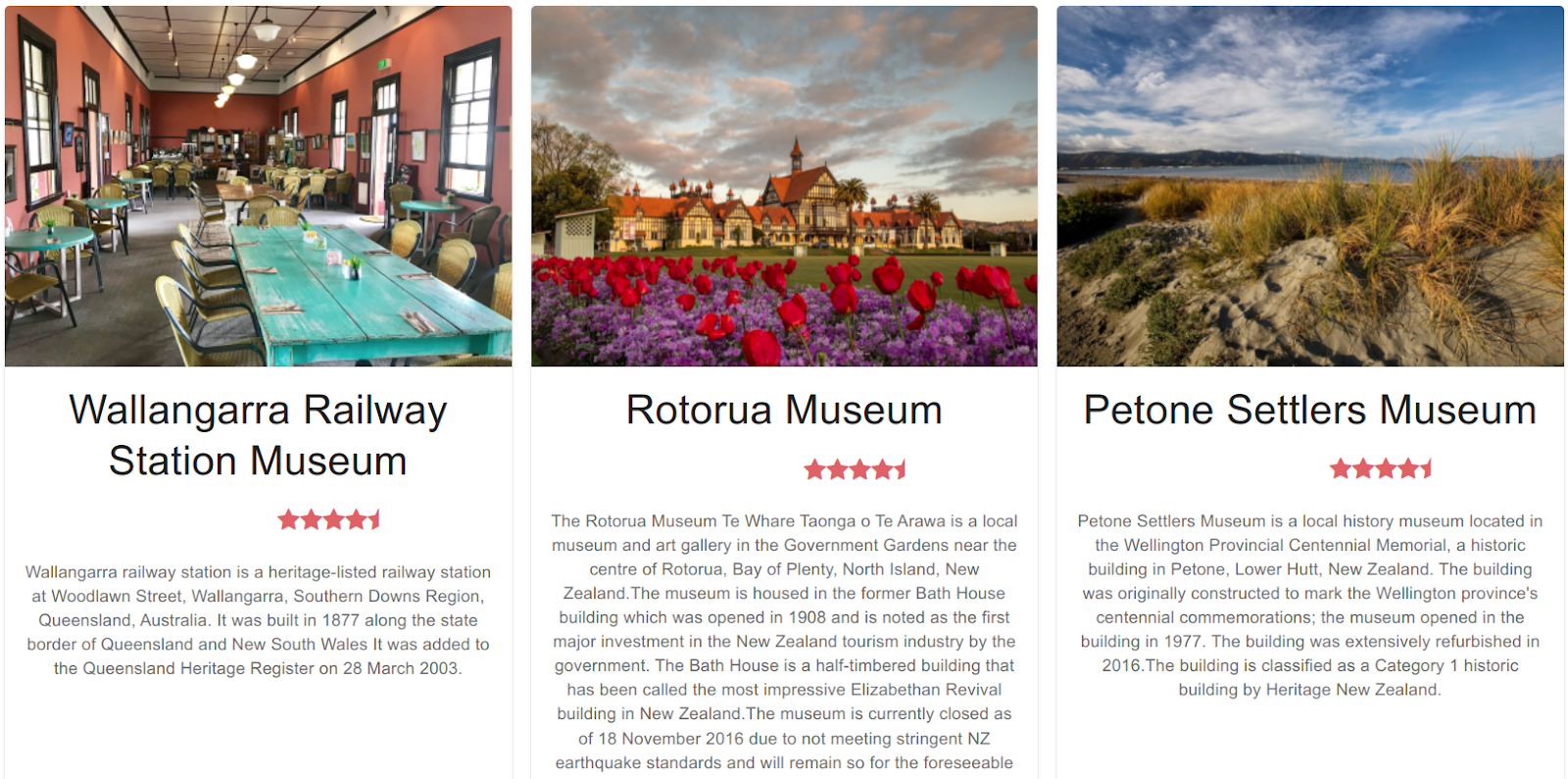
If we get other attractions only using the types of this car museum which involve “cultural”, “museum”, “museum of science and technology”, museums that are irrelevant to transportation will be returned.

*Visual Feature Only w/o Type Refinement*

**

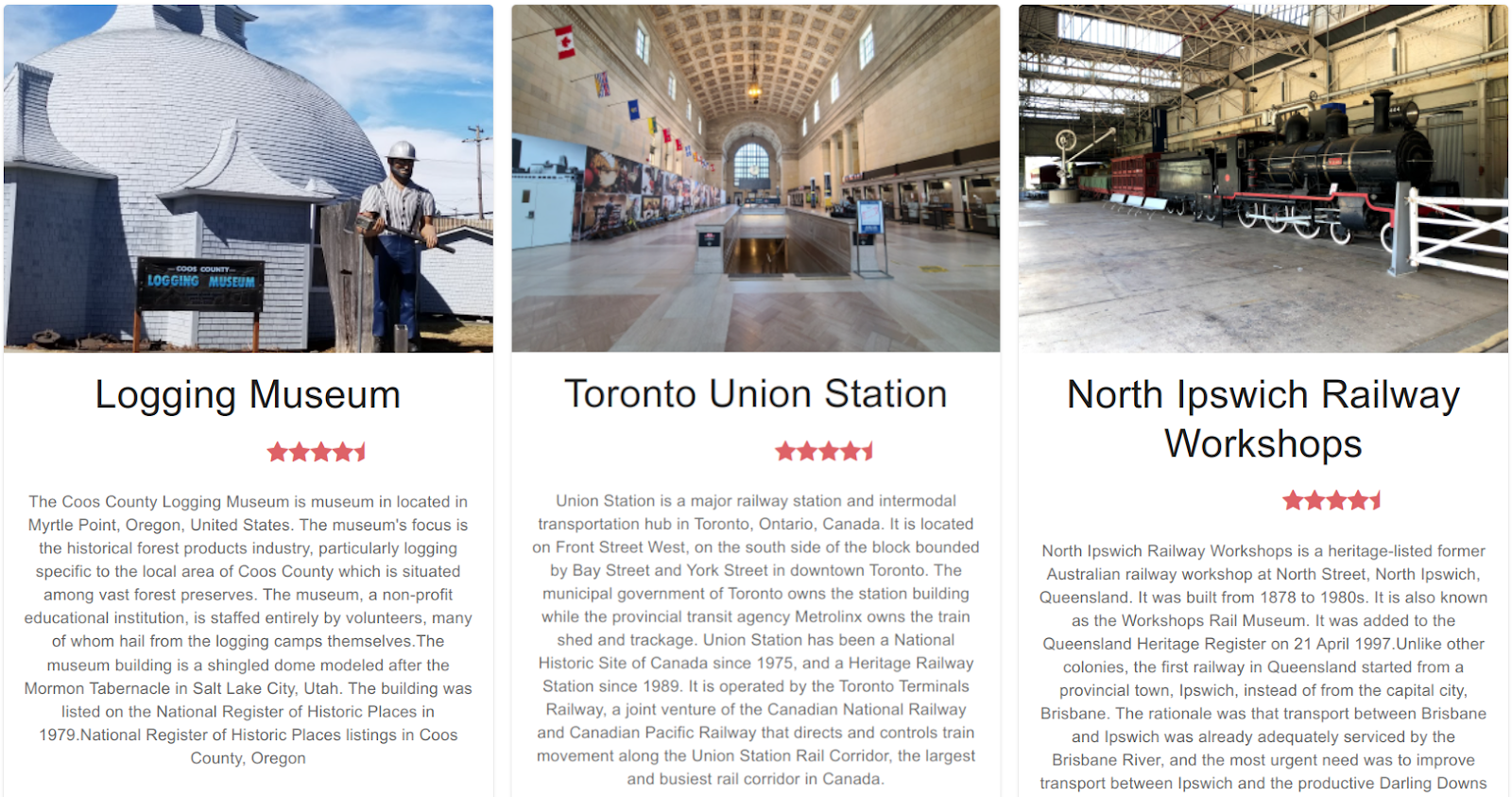
Pictures of the returned results contain cars, but they are not really museums.

*Language Feature Only w/o Type Refinement*

**

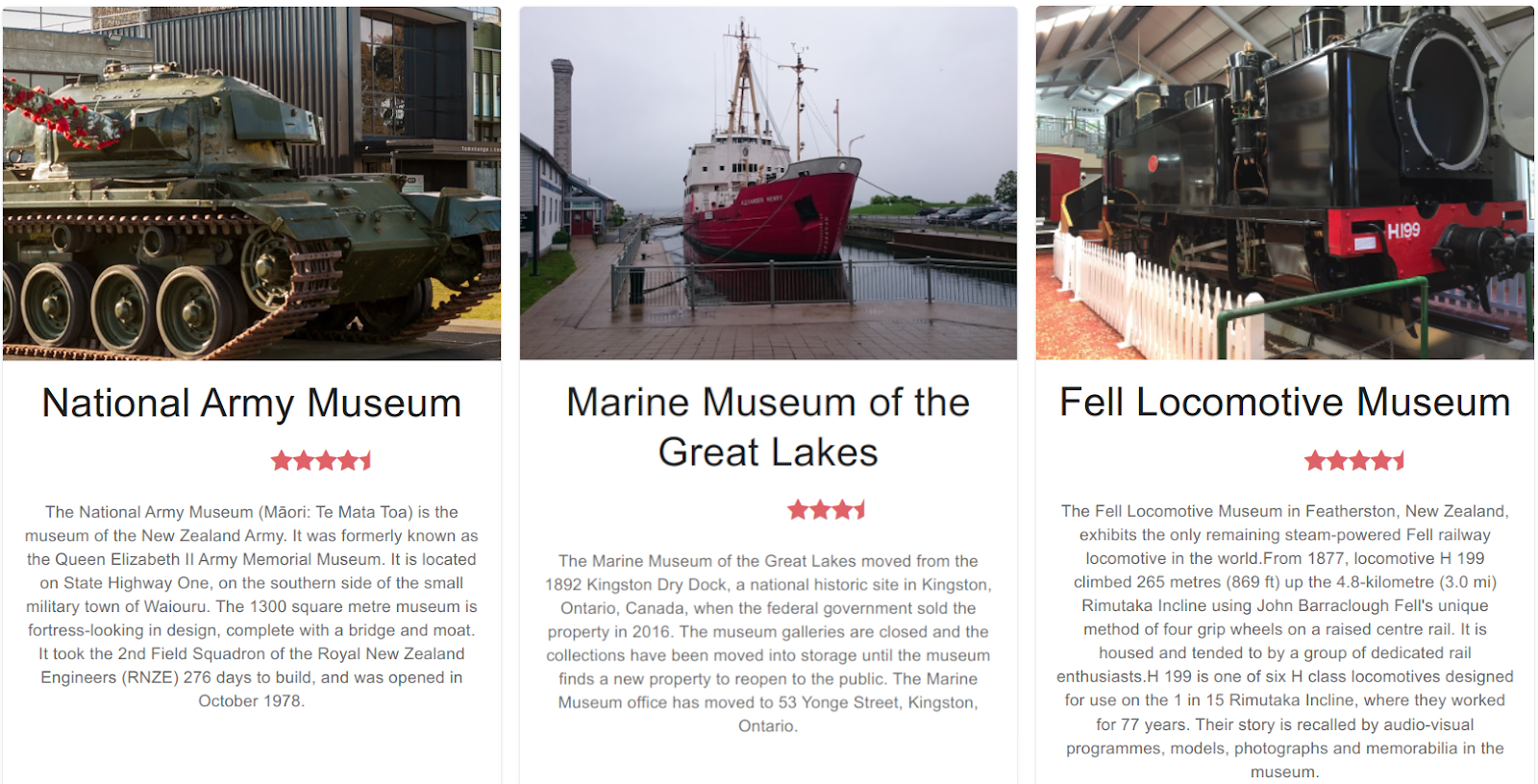
The descriptions returned contain “museum” (some also contain New Zealand), yet visually these places are not quite related to transportations.

*Visual+Language Features w/o Type Refinement*



The results returned are related to either museum or transportation (not necessarily both).

*Visual+Language Features with Type Refinement*

**

The results contain museums exhibiting vehicles or transportations which make more sense.

In conclusion, combining visual and language features with type refinement outperforms the baselines of type-only, visual-only and language-only approaches.