6 Implementation and Deploy

6.1. Implementation Issues

Analyzing the architecture described above, there are still some critical points related to scalability. One is the data repository or storage of item sets, since the greater the number of items and amount of feedback are, the greater is the amount of data required to represent them. Another critical issue related to data access is the time required to read and write information, which can lead to request queuing as the number of nodes in the system increases. A third area of attention is the queues used to connect the different layers, which may limit the number of reads and writes, and cannot scale as the number of operations per second increases, for example, when there is a burst of user feedback.

To address problems related to the data repository, the proposed architecture uses a distributed in-memory representation, which isolates read operations from write operations. Existing solutions that use system memory to store data for faster access include Membase [61], Memcached [62], Hazelcast [63], and Redis [64]. We chose Redis as it comes with built-in data structures that are particular suited to the application in question: sets, sorted sets and lists. Additionally, Redis natively implements an intersection operation between sets, which can be used directly for the calculation of the dot product between the item vectors, as described above.

Another advantage of Redis is the existence of sorted sets. The representation of a similarity graph can be implemented using this structure, where each item corresponds to an ordered set of other items, with the sorting factor being the similarity between them. The use of Redis for similarity data storage allows the retrieval of similar items to be done in constant (fixed) time.

In addition, by using Redis as a data repository, it is possible to store the queues within it, as the implementation allows for the inclusion and exclusion of items in constant time. Redis holds up to 200,000 queue operations per second [64].

Finally, and more importantly, it is possible to set a life span for keys (sets, sorted sets, lists, etc.). This is particularly interesting for the recommendation engine, as it allows the inclusion of temporal dynamics in our solution, without the need to develop manual controls. In a nutshell, keys expire when their pre-determined lifespan is reached, and they are subsequently purged from the dataset. Thus, the introduction of temporal dynamics in our solution is achieved by setting an expiration time for each piece of feedback, which should reflect the period of time that the feedback is relevant to the recommendation model. After this expiration time, items are purged and no longer considered in the similarity calculation.

Figure 19 illustrates how Redis pools are integrated in the proposed architecture.

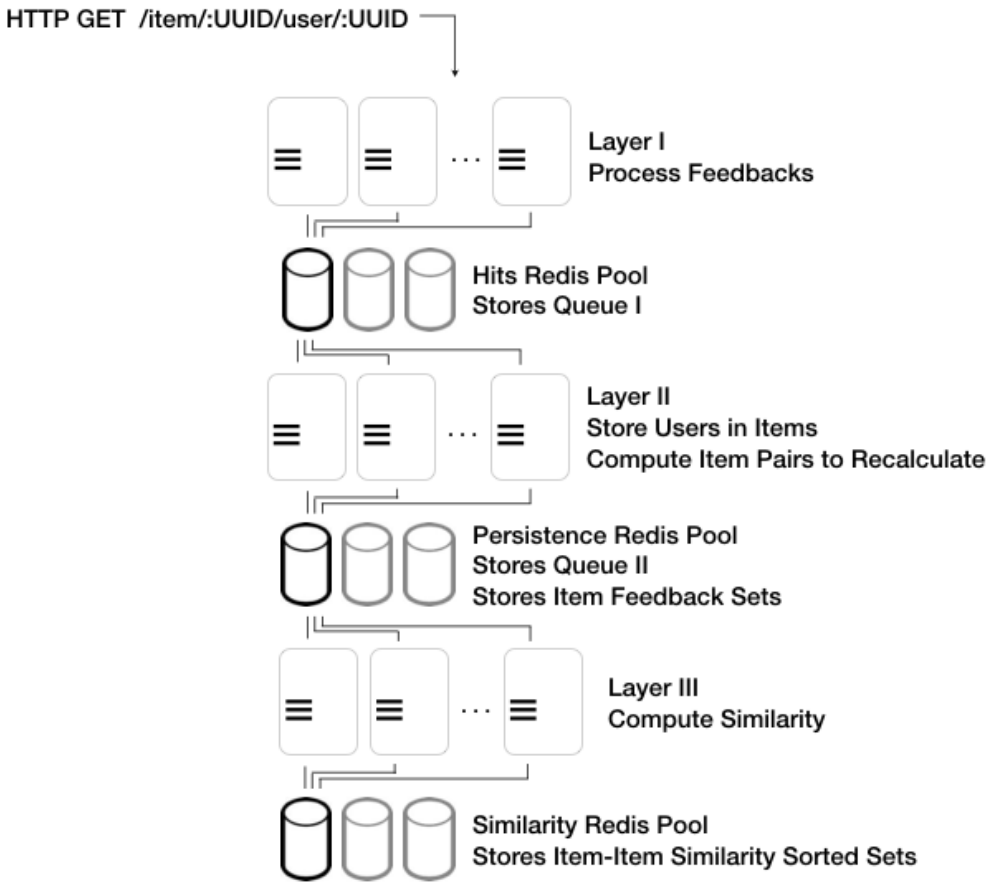

Figure 19 - Redis pools connecting architecture layers

As shown, besides the isolation of processing layers, there is also an insulation of data repositories, since each one has different requirements with respect to computing resources, amount of memory to represent the information, and the

number of read/write operations. Thus, we can use the available resources of an IaaS platform more efficiently, which ultimately means obtaining the recommendation in less time and at a lower financial cost.

With the architecture layers isolated, it is possible to scale each one independently and on-demand. Thus, monitoring agents were developed to create or destroy server instances as needed. As mentioned, for the front-end (FE) servers that receive requests from users, resource consumption and response times are monitored to scale up or down consumption. In the other two layers, the monitor agent probes the processing queues size to create or destroy worker nodes with the aim of ensuring that queues are always empty, so that a recommendation can be obtained in real-time.

Another important consideration is fault tolerance. With the layers isolated and the processing distributed, each node performs one operation at a time, initiated by the removal of an element from a queue, which takes place in constant time. Thus, the failure of any of the workers only means the loss of that work, which is not critical to the reliability of the recommendations made. Moreover, in the case of a node failure, the agents can start another one, as needed. Failures can easily be identified by the growth in the number of items in the queues. Regarding monitoring agent failover, these agents are executed in parallel on multiple servers and share information through a dedicated instance of Redis, also redundant, which ensures constant monitoring.

6.2. Deploy using Amazon Web Services

To validate the proposed architecture, it was deployed on the Amazon Web Services (AWS) cloud computing platform [3]. The reasons for choosing this particular vendor are the adherence to Public Cloud IaaS Criteria (Amazon meets 71% of Gartners required production-grade enterprise IaaS criteria) and the fact that AWS is the market leader with offices in Brazil [65, 66].

The first step in this process involved configuring the servers and their respective roles and creating images (Amazon Machine Images - AMIs) for each role to enable a rapid instance startup based on these images, according to demand fluctuations. Thus, images were created for: (1) the front-end servers, comprising a web server using Nginx with a Redis2 module, and able to write directly to the queue

connecting the first two layers, (2) the nodes of the second layer, which consume the data in this first queue and structure the data based on the sets representing each item, (3) the nodes in the third layer, which calculate the similarity between items, (4) the servers that run the master and slave Redis instances, responsible for data storage, and (5) the servers that run the monitoring agents.

Once images had been created and servers configured, the load balancing service, Elastic Load Balancer (ELB), was employed to enable better scalability of user requests, and also to isolate access to data repositories, allowing the startup or shutdown of instances without any environment reconfiguration needed. Figure 20 shows the final architecture deployed on AWS.

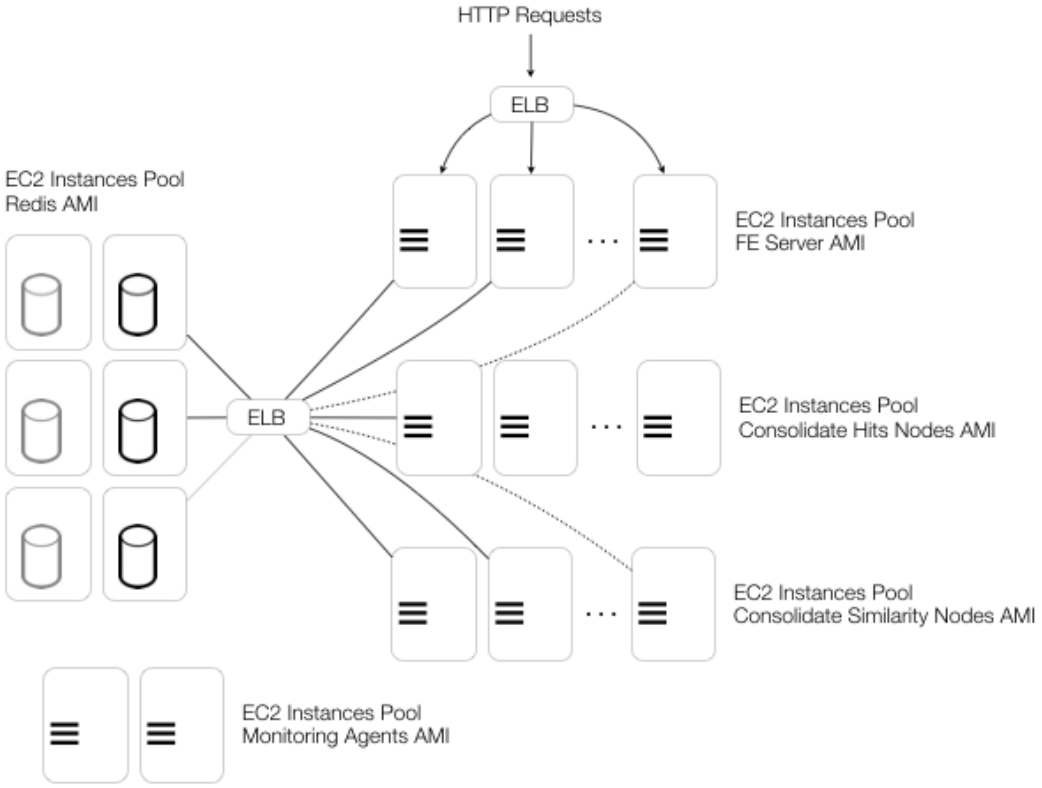

Figure 20 - The recommendation platform deployed on AWS

Note that any element of the architecture can be scaled up or down to meet demand, thus enabling complete scalability, which is crucial for dynamic environments with great variations in the numbers of users and items.