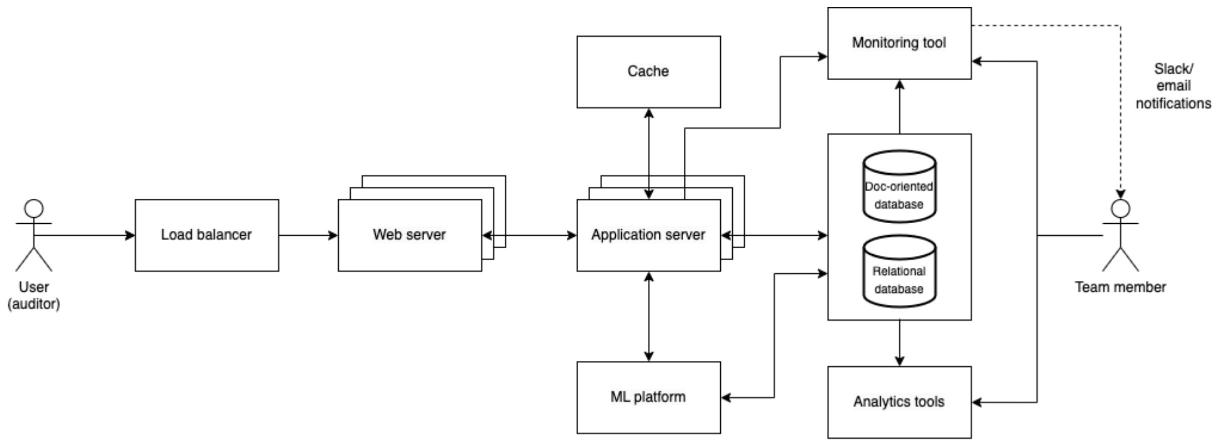
# OpenZeppelin ー ML Engineer Work Test

## Part II - System Design

This document describes the system design of the contract similarity finder. The goal of this tool is to help auditors to find similar audited smart contracts and aid their current audit by reliably and accurately finding similar contracts.

The system must meet the following requirements:

- ML training data collection and retraining
- ML model monitoring for performance regression
- Feedback loop that'll take in auditor's feedback and store for future ML retraining
- Return top k similar contracts with < 1 second latency



Schema of the proposed system design

### User (auditor)

The user initiates the request to find similar Solidity contracts by opening a web browser, navigating to the URL where our tool is available, copy-pasting in the currently audited code, the count of similar contracts they want to receive, and hitting search.
After the similar contracts have been returned, they are displayed to the user in a UI similar to VS Code with the screen split into two parts, the audited contract on the left, and the returned contract on the right with the possibility to click through the returned contracts in the upper panel. After reviewing the returned contracts, the user then has the option to reissue the same search with an increased number of similar contracts, or provide feedback on the individual returned solidity files by giving a thumbs up or a thumbs down to any of the returned contracts.

### Load balancer

The load balancer (e.g. Nginx) is placed between the client and the web server to distribute the requests to multiple instances of the web server. By distributing the workload, we can achieve a faster user experience and prevent the web server from being a single point of failure, which would lead to the entire system being inaccessible.

The load balancing technique used would be the least connection method as we can expect the individual connections to take some time. This approach also allows us to account for the different parameters and computational speed of individual servers and differences in duration of the most similar contracts search.

## Web server

The web server is responsible for responding to HTTP requests made by clients. First, it returns the web page that serves as the UI of the similar contract tool, then when a particular search is initiated, it processes the request and passes the search parameters onto the application server. The application server then executes the required logic and sends the response back to the web server, which returns it to the client.

## Application server

After the application server receives a request from the web server, it first tries to retrieve the most similar solidity files to the given source code from the cache. In case of a cache hit (representing the situation where an auditor has queried the app for more similar contracts that were returned previously), the cache sends the next similar contracts to the application server, which then passes them back to the auditor. This approach helps to mitigate querying the same smart contract multiple times. In case of cache miss (representing the first query for one smart contract), the application executes an API call to the appropriate ML platform endpoint to search for the most similar contracts using the solidity files stored in database, and then stores the result (the for example 50 most similar contracts together with the queried contract) into the external cache for faster retrieval in the future. The total number of similar contracts that will be retrieved and stored in the external cache should be chosen as the upper bound of how many similar contracts the auditors might request and make use of.

The application layer also contains a daemon continuously updating the solidity contract database. The daemon fetches either the list of last verified contracts, or new Ethereum blocks, and uses the Etherscan's contract API endpoint to obtain the code of the recently verified contracts. Subsequently, these recently verified contracts are appended to the database and marked to be processed by the machine-learning models.

## Cache

Cache is connected to the application server and is designed using the cache-aside strategy. It returns the results from previous searches if possible and if not, returns a cache miss response and stores the results once they have been computed and sent by the application server. In case there is not enough space in the cache, the LRU (least recently used) eviction strategy is applied to maximize the probability of a cache-hit in the event of a follow-up request for more similar contracts.

## Database

The database part of the system is designed to consist of two separate databases. One is a document-oriented database (for example MongoDB) that contains all verified solidity contracts that are used for retrieval when they are identified as the most similar to the currently examined document provided by the auditor, and separately the contracts sent by auditors linked to the auditor via a session ID and a hash of the sent contract to allow for one auditor querying several contracts during one session. The advantages of using a

document-oriented database for this data is that it's suitable for unstructured data, quick to set up, easy to use, and has high scalability by adding more nodes to the cluster, which is useful as the total number of verified contracts increases every day.
The second database is a relational database (for example MySQL) and has three tables - one stores the transaction hashes of the deployed contracts, their location in the document-oriented database, and embedded vectors created from the source code for faster similarity computations. The second table contains the same information as the previous one except for the embeddings. This is the table that the database update daemon appends to and the ML platform then reads from and processes the contracts in it. The last table persists the user's feedback by storing the session ID of the auditor providing feedback, the hash of contract that was sent by them, the transaction hash of the contract the feedback is given for, and the feedback value (either positive, or negative), and a date when the feedback was given.

During set-up of the system, we would obtain and upload a large dataset of solidity contracts to the database. For this, we could use some publicly available datasets of audited solidity smart contracts, for example a CSV list of verified contracts addresses of which the code publishers have provided a corresponding open source license for redistribution followed by obtaining the source codes of these contracts via the contract API endpoint of Etherscan.

To achieve scalability, the data in the document-oriented database could be partitioned into different database servers using the first letter or two of the transaction hash would be used for partitioning. For the relational database, we could partition the documents in the same way, and the feedback records could be partitioned by the date if we would expect similar amounts of feedback per day over time, or also by the first character of the transaction hash of the returned contract, which should lead to a more even distribution over time. On the other hand, for the purpose of retraining, partitioning by date is more useful and faster.

## Machine-learning platform

A machine-learning platform (e.g. Amazon SageMaker) is where the machine-learning models reside. It provides data preprocessing, contract embedding, model training and retraining based on the provided feedback. The model is automatically periodically retrained based on new inputs from the users and then deployed as the new version of the model. The similarity model is available via several concurrent API endpoints from the application servers. The API call contains the examined contract code, which is then processed and converted to an embedded, vector representation inside the ML platform. Subsequently, the search for the most similar contracts is executed and the API returns the transaction hashes of the 50 most similar contracts back to the application server.
Given the low expected amount of user feedback at first, the model would be augmented using the feedback once a week.
The contracts would be transformed into word embeddings by using a pre-trained model, such as Word2Vec or GloVe. The use of pre-trained word embeddings can improve the performance of the model and reduce the amount of training data required.
The machine-learning model used for computing the contract similarity could be a variant of a Siamese neural network. The Siamese network architecture is well suited for similarity comparison tasks as it can take two input sequences and output a similarity score between them. The Siamese network architecture would be built using either TensorFlow or Pytorch.

To implement the reinforcement learning component of the system, we could use Q-learning, a model-free reinforcement learning algorithm that can learn to make optimal decisions based on trial and error. Q-learning would be used to augment the Siamese network based on the feedback provided by the auditors, and the model would be also retrained periodically (daily) to incorporate recently released contracts.

## Monitoring tool

A monitoring tool, such as Grafana, is set up to track the system's performance in real-time, most importantly latency, error rates, and throughput. Additionally, it displays dashboards providing more insights and trends regarding the contract similarity tool, for example count of daily usage over time, and count and sentiment of auditor's feedback. Alerts are set up for team-members to receive via Slack and/or email in case of some set threshold is exceeded, for example if the contract similarity tool's response time exceeds a certain threshold or if the ratio of the auditor's feedback over the past 24 hours is 80% or more negative.

## Analytics tools

The auditor's feedback can be further examined and analyzed using analytics tools such as Tableau. We can search for any common patterns that are linked to downvoting returned contracts, or identify contracts that receive high ratios of downvotes and might be worth removing from the database or retraining the model.

## Summary

Overall, this system design will enable us to reliably and accurately find similar documents while supporting the tool to return top k similar contracts with minimum latency. The ML training data collection and retraining process, ML model monitoring for performance regression, and feedback loop for auditor's feedback will help ensure that the system remains accurate and reliable over time.
If the database has been set up with enough redundancy having more than one replica of the data, the system should only have one bottleneck and that is the machine-learning platform. In case of a failure of one of the API endpoints, the others can handle the increased workload, however, in case of the entire platform going down, the system would not be able to operate correctly. This could be solved by creating a back-up solution, such as having a copy of the ML model inside of the application servers that would be updated once a week and could serve the requests in the case of the server being unresponsive.
If we wanted to further increase the computational speed of the similar contract detection part, we could apply some clustering or segmentation to the contracts available. Either based on the nature of the contract (NFTs, DeFi, etc.), or purely based on the vector representation of the contracts, we could cluster the contracts into several groups, choose one example of each group (usually the average or the one closest to the average of all samples) and then start the find by only comparing the audited document with the representatives of each cluster. After finding the cluster that is the closest, we could then execute the search only within the specific cluster instead of comparing the contract to all in the database.