Orissa, India

+91 8895319373

Somes.dash1995@gmail.com

**Cryptocurrency Liquidity Prediction for Market Stability**

Low Level Design

Domain: Machine Learning

Creator: Somes Dash

Date: 22.05.2025

|  |  |  |  |
| --- | --- | --- | --- |
| Date issued | Version | Description | Author |
| May 22, 2025 | 1.1 | First Draft | Somes Dash |

**Document Version Control**

Contents

[Introduction 3](#_Toc110433758)

[What is Low-Level Design Document? 3](#_Toc110433759)

[Scope 3](#_Toc110433760)

[Architecture 4](#_Toc110433761)

[Architecture Description 5](#_Toc110433762)

[Data Preparation 5](#_Toc110433763)

[Data Description 5](#_Toc110433764)

[Data Preprocessing 5](#_Toc110433765)

[Exploratory Data Analysis 5](#_Toc110433766)

[Feature Engineering 5](#_Toc110433767)

[Model Development 6](#_Toc110433768)

[Model implementation 6](#_Toc110433769)

[Hyper-parameter Tuning 6](#_Toc110433770)

[Model Evaluation 6](#_Toc110433771)

[Deployment 6](#_Toc110433772)

[Designing UI with Anvil 6](#_Toc110433773)

[Designing a server 6](#_Toc110433774)

[Code deployment on cloud 7](#_Toc110433775)

[Deployment Process 7](#_Toc110433776)

# Introduction

## Here’s a version adapted for your cryptocurrency liquidity prediction project:

## **What is a Low-Level Design Document?**

## The Low-Level Design (LLD) document provides a detailed internal view of the actual program code and logic behind the Cryptocurrency Liquidity Prediction system. It explains the system’s purpose, key features, internal modules, interfaces, and the expected behaviour under various conditions. The LLD defines how different components interact, the constraints they operate within, and how the system responds to external inputs.

## The main objective of this project is to accurately predict liquidity levels (low, medium, high) for cryptocurrencies on specific dates by analysing critical market indicators such as price changes, trading volume, market capitalization, and other derived features. This detailed design guides developers in implementing the system’s functionalities and ensures alignment with the overall project goals.

## Scope

Low-Level Design (LLD) is a detailed, component-level design approach that involves step-by-step refinement of the system’s architecture. In this project, LLD focuses on designing data structures, defining the software components, implementing source code, and optimizing the performance of the liquidity prediction algorithms. While the overall data organization and requirements are identified during the initial analysis phase, the LLD phase refines these aspects to create a clear blueprint for development. This ensures that all components—from data processing to machine learning model integration—are precisely defined and aligned with the project goals.

Architecture

**Data Preparation**

**Model**

**Development**

**Deployment**

# Architecture Description

## Data Preparation

### Data Description

Hourly Interstate 94 Westbound traffic volume for MN DoT ATR station 301, roughly midway between Minneapolis and St Paul, MN. Hourly weather features and holidays included for impacts on traffic volume. The goal of this project is to build a prediction model using multiple machine learning techniques and to use a template to document the end-to-end stages. We're trying to forecast the value of a continuous variable with the Metro Interstate Traffic Volume dataset, which is a regression issue.

**Architecture Description**

**Data Preparation**

Data Description

The dataset used for this project contains daily-level information on various cryptocurrencies. It includes essential metrics capturing price movements, trading activity, and market capitalization. The original dataset comprises the following columns:

* **coin**: Name of the cryptocurrency (e.g., Bitcoin, Ethereum).
* **symbol**: Ticker symbol representing each cryptocurrency (e.g., BTC, ETH).
* **price**: The current trading price of the cryptocurrency on a given date.
* **1h, 24h, 7d**: Percentage change in price over the last 1 hour, 24 hours, and 7 days, respectively—these features capture short-term and medium-term price volatility.
* **24h\_volume**: The total trading volume over the past 24 hours, reflecting market activity and liquidity.
* **mkt\_cap**: Market capitalization of the cryptocurrency, indicating its market value based on the circulating supply and current price.
* **date**: The corresponding date of the recorded data.

Data Preprocessing

* The dataset was examined for missing or null values across all columns.
* Any rows with missing essential numerical data (such as price, volume, or market cap) were dropped to maintain data integrity.
* Dates were formatted uniformly to ensure proper time-series ordering.

Feature Engineering

To enhance the predictive power of the models, several new features were engineered from the original dataset:

* **Lag Features**:
  + price\_lag1, volume\_lag1, and mktcap\_lag1 represent the values of price, 24h volume, and market capitalization from the previous day.
* **Rolling Averages**:
  + price\_2d\_avg and volume\_2d\_avg are two-day moving averages used to smooth out short-term fluctuations.
* **Ratio-Based Features**:
  + vol\_to\_mcap: Ratio of 24h volume to market capitalization.
  + vol\_price\_ratio: Ratio of 24h volume to current price, indicating trading intensity.

**Log Transformation**

To handle the large magnitude differences and skewed distributions commonly found in financial data, logarithmic transformation was applied to selected features:

* log\_price = log1p(price)
* log\_volume = log1p(24h\_volume)
* log\_mkt\_cap = log1p(mkt\_cap)

This transformation:

* Compressed the scale of the data.
* Reduced the effect of extreme values (outliers).
* Helped stabilize variance and improve the performance of machine learning models by making the data more normally distributed.

**🔹 Lag Features (Time Dependency)**

In order to incorporate time-based patterns and temporal dependencies, **lag features** were created for each cryptocurrency (symbol) using the group-by technique:

grouped = df.groupby('symbol')

df['price\_lag1'] = grouped['log\_price'].shift(1)

df['volume\_lag1'] = grouped['log\_volume'].shift(1)

df['mktcap\_lag1'] = grouped['log\_mkt\_cap'].shift(1)

These lag features represent the previous day’s log-transformed price, volume, and market capitalization for each coin. They are crucial in modeling sequential trends and temporal behavior in the market, especially for models that capture time dynamics like LSTM.

* price\_lag1: Helps capture short-term price momentum or reversals.
* volume\_lag1: Helps identify surges or drops in trading activity.
* mktcap\_lag1: Helps track shifts in investor interest and market valuation over time.

The inclusion of these lag features enabled the models to better understand historical dependencies, thus enhancing their predictive accuracy for both regression and classification tasks.

**Exploratory Data Analysis (EDA)**

Exploratory Data Analysis (EDA) was performed to understand the structure, patterns, and relationships in the cryptocurrency market data. This step helped in identifying key trends, detecting anomalies, and guiding the feature engineering and modeling process.

**Summary Statistics**

* Descriptive statistics (mean, median, standard deviation, min, max) were calculated for all numerical columns such as price, 1h, 24h, 7d, 24h\_volume, and mkt\_cap.
* The data showed high variability in trading volume and market capitalization, reflecting the volatile nature of the cryptocurrency market.

**Distribution Analysis**

* Histograms and KDE plots were generated to examine the distribution of key variables:
  + price and volume were right-skewed, with a few coins having extremely high values.
  + After applying log1p transformation on 24h volume, the distribution became more symmetric and suitable for modeling.

**Correlation Matrix**

* A correlation heatmap revealed:
  + Strong positive correlation between 24h\_volume and mkt\_cap, indicating larger market cap coins tend to have higher trading volume.
  + Price change percentages (1h, 24h, 7d) showed weaker correlations with liquidity, suggesting that short-term price movement alone isn't a strong predictor.

**Temporal Patterns**

* Time-series plots for top cryptocurrencies highlighted fluctuations in price and volume over time.
* Some coins showed periodic spikes in volume, possibly due to news events or market sentiment shifts.

**Liquidity Level Distribution**

* The liquidity\_level target variable was created using quantiles and categorized into low, medium, and high.
* A bar plot showed a balanced distribution across the three liquidity classes, confirming that quantile binning effectively divided the dataset without severe class imbalance.

**Outlier Detection**

* Boxplots were used to detect outliers in features like price, volume, and mkt\_cap.
* A few outliers were present, particularly for smaller coins with highly volatile trading patterns. However, these were retained as they represent realistic market behavior.

**Pairwise Relationships**

* Pair plots helped visualize pairwise relationships between selected features, confirming some linear and non-linear interactions, which justified the use of ensemble and deep learning models.

This EDA provided critical insights that shaped the feature engineering strategy and informed model selection. Visualizations and statistical summaries helped validate the quality and predictive potential of the data.

**Model Development**

**Hyperparameter Tuning**

To enhance the performance of the Random Forest classifier and prevent underfitting or overfitting, hyperparameter tuning was conducted using Grid Search with Time Series Cross-Validation.

🔹 Tuning Strategy

* Model: RandomForestClassifier
* Search Method: GridSearchCV
* Cross-Validation Type: TimeSeriesSplit with 5 splits, preserving temporal dependencies in the data (important for time series forecasting tasks).
* 🔹 Parameters Tuned

The following hyperparameters of the Random Forest model were tuned:

| Parameter | Values Tested |
| --- | --- |
| n\_estimators | [100, 200] |
| max\_depth | [10, 20, None] |
| min\_samples\_split | [2, 5] |
| min\_samples\_leaf | [1, 2] |

This resulted in a total of 24 combinations being tested across the 5 cross-validation folds.

🔹 Performance Metric

* Scoring Criterion: accuracy — chosen as the primary metric for assessing model performance during classification.
* Computation: Parallelized using n\_jobs=-1 to speed up the process.

🔹 Results

After performing the grid search, the following results were obtained:

* Best Hyperparameters:
* print("Best RF Parameters:", grid\_rf.best\_params\_)
* Cross-Validation Accuracy:
* print("Best RF CV Score:", grid\_rf.best\_score\_)

These optimized parameters were then used to refit the best estimator, which was later used to generate class probability predictions for the stacking ensemble.

🔹 Importance of TimeSeriesSplit

Using TimeSeriesSplit instead of standard k-fold cross-validation is critical in time-dependent datasets like cryptocurrency trading data. It ensures that training always precedes testing chronologically, preventing lookahead bias and ensuring a more realistic evaluation.

**Model implementation**

To address the problem of predicting cryptocurrency liquidity levels (classified as **High**, **Medium**, or **Low**), a **stacked ensemble classification approach** was implemented, leveraging the strengths of both traditional machine learning and deep learning models.

**🔹 Problem Framing**

The target variable, **liquidity level**, was derived based on log-transformed 24h volume (log\_volume) and categorized into three discrete levels. This framed the task as a **multi-class classification problem**.

**🔹 Data Preparation for Modeling**

**🔹 Features Selected**:  
 A set of engineered features was selected based on temporal dependencies and market indicators:

* + '1h', '24h', '7d': Price change percentages over short durations.
  + 'price\_lag1', 'volume\_lag1', 'mktcap\_lag1': Lagged values representing time-dependence.
  + 'price\_2d\_avg', 'volume\_2d\_avg': Recent 2-day averages.
  + 'vol\_to\_mcap', 'vol\_price\_ratio': Derived ratios providing relative market dynamics.
* **Label Encoding**:  
  The categorical target liquidity\_level was encoded using LabelEncoder() and further transformed into one-hot encoded vectors for LSTM input.
* **Scaling**:  
  All features were standardized using StandardScaler to ensure consistent scaling across models.
* **Sequential Input Creation**:  
  For LSTM, input sequences were generated using a sliding window of size 1 (time\_step=1), maintaining temporal order without shuffling.

**🔹 Base Models**

1. **Random Forest Classifier**  
   A robust ensemble of decision trees was trained on flattened feature vectors (x\_train\_flat). It captured non-linear interactions without needing time dependencies.
2. **LSTM Classifier (Long Short-Term Memory)**  
   A two-layer LSTM network was designed to model time-dependent behavior in sequential data. The network ended with a dense output layer using softmax for multi-class classification.
   * Architecture:
     + LSTM(64, return\_sequences=True)
     + LSTM(32)
     + Dense(3, activation='softmax')
   * Loss: categorical\_crossentropy
   * Optimizer: Adam
   * Evaluation Metric: Accuracy

**🔹 Meta-Model (Stacking)**

A **meta-classifier** was introduced to combine the predictive strengths of both base models:

* **Input**: Probability predictions (predict\_proba) from both Random Forest and LSTM.
* **Model Used**: Logistic Regression
* **Goal**: Learn optimal combination of the base predictions to maximize classification performance.

The best model is chosen, and Grid Search with Cross Validation is applied on that model to get the best parameters. Those parameters is then used on the model to get better result.

**Model Evaluation**

Each model was evaluated using:

* **Classification Report**: Precision, Recall, and F1-score for each liquidity class.
* **Accuracy Score**: Over all classification accuracy.

**Model Saving**

To facilitate model reuse, reproducibility, and deployment, all trained models and preprocessing objects were saved after training using appropriate formats

**🔹 Scikit-learn Models**

The following models and tools were saved using joblib, which is efficient for serializing Python objects:

* **Random Forest Classifier**:  
  Saved as random\_forest\_model.pkl — the best estimator from GridSearchCV.
* **Meta Classifier (Logistic Regression)**:  
  Saved as meta\_model.pkl — the stacking model used to combine predictions from Random Forest and LSTM.
* **Feature Scaler**:  
  Saved as scaler.pkl — used to standardize input features during preprocessing.
* **Label Encoder**:  
  Saved as label\_encoder.pkl — used to encode target class labels (low, medium, high liquidity levels).

**🔹 LSTM Model**

The LSTM deep learning model was saved in **HDF5** format using Keras’s built-in method, ensuring the model architecture, weights, and optimizer state are all preserved.

These saved models and preprocessing tools can later be **loaded into a deployment pipeline** for inference on new cryptocurrency data or integrated into an application for real-time liquidity prediction.

**Model Deployment**

After developing and evaluating the machine learning models for cryptocurrency liquidity prediction, the deployment phase involved creating a web application to provide real-time liquidity forecasts to users.

The backend API was built using **Flask**, a lightweight Python web framework, to serve the trained models and handle prediction requests. The Flask app (app.py) was developed locally in **Visual Studio Code (VS Code)** and version controlled using **Git**. The project repository was pushed to **GitHub** for source code management and collaboration.

For hosting and serving the application online, the codebase along with the trained models and necessary dependencies were deployed on **Render**, a cloud-based platform that offers easy deployment and scalable hosting for web applications.

This deployment setup enables users to access the model’s predictions via a user-friendly interface in real time, making the liquidity prediction tool readily accessible from anywhere with an internet connection. The combination of Flask, GitHub, and Render ensures efficient development workflow, version control, and reliable cloud deployment.

# Unit cases

**Test Case Description**

| **Test Case** | **Pre-Requisite** | **Expected Result** |
| --- | --- | --- |
| Verify whether the application URL is accessible to the user | Application URL should be defined | Application URL should be accessible to the user |
| Verify whether the application loads completely when the URL is accessed | 1. Application URL is accessible2. Application is deployed | The application should load completely for the user |
| Verify whether the user is able to see all input fields | Application is accessible | User should be able to see all input fields |
| Verify whether the user is able to edit all input fields | Application is accessible | User should be able to edit all input fields |
| Verify whether the user is provided with a Submit button to submit inputs | Application is accessible | User should see a Submit button to submit the inputs |
| Verify whether the user is presented with results on clicking Submit | Application is accessible | User should be presented with results upon submission |
| Verify whether the results correspond to the user’s inputs | Application is accessible | Results should accurately reflect the inputs submitted |