Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Dynamic Virtual Automated Market Makers and their limitations in Decentralized Finance

Aditya Kumar Shrivastava

April 24, 2022

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MAI (Computer Engineering)

Declaration

I hereby declare that this dissertation is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed: _____ Date: _____

Abstract

Decentralized finance, or DeFi, is an area of financial services provided and operated on a blockchain. The total trading volume in decentralized exchanges or DEXs exceeds $4 Billion daily. Derivative trading has grown rapidly in the DeFi ecosystem and is one of the most popular DeFi services, with the total liquidity locked in derivatives trading being over $1.8 billion. The DeFi ecosystem has much liquidity locked in derivatives trading, especially in perpetual swap contracts. Perpetual swaps are derivatives that let a trader speculate on the value of an asset. The algorithms and protocols that currently facilitate the trading of perpetual contracts are highly innovative; however, they face several problems. A major problem faced by the existing protocols is their insensitivity and failure to adapt to external market conditions and market volatility for an underlying asset. This project aims to explore, design, develop, and evaluate a mechanism that will enable an existing protocol to adapt to external market conditions for any asset speculated by a perpetual contract. This study specifically explores and implements a mechanism to configure existing virtual automated market-making protocols to allow them to adapt to external market conditions. This project designs and implements a configurable or dynamic virtual automated market maker (vAMM) protocol for a perpetual swap exchange. The configurable vAMM is built on top of a stock vAMM protocol design. Experiments are performed on the implemented configurable vAMM protocol to evaluate if the implemented protocol can solve the existing problem of market adaptation with the vAMM protocol. The evaluation of the developed configurable or dynamic vAMM protocol showed that the protocol could adapt to external market conditions. However, the evaluation also showed that the protocol is not a viable solution and introduces critical issues that disrupt the core functionalities of a perpetual swap exchange. The implemented protocol needs additional mechanisms to be developed to solve its limitations. It is found that addressing these limitations of the implemented protocol can render the dynamic vAMM protocol a viable solution. This study also discovers the impact of key parameters of an automated market-making protocol on perpetual swap exchanges and discusses the importance of these parameters. Finally, this research suggests an algorithm that would automate the process of configuring a vAMM protocol to help a perpetual swap exchange adapt to market conditions.

Acknowledgements

I would like to thank my supervisor Dr Donal E. O'Mahony, for his constant invaluable guidance and mentorship. This research project would not have been possible without his support.

I would like to thank the module leader for the Computer Engineering Research Project, Ms Paula Roberts, for her continuous support throughout this project.

Lastly, I would like to thank my parents for their assistance in understanding financial markets and for their continuous moral support.

Contents

List of Figures

List of Tables

Nomenclature

AMM	Automated Market Maker	
BTC	Bitcoin	
DEX	Decentralized Exchange	
DvAMM	Dyanmic virtual Automated Market Maker	
ETH	Ether token of the Ethereum Blockchain	
USDC	US Dollar (USD) Coin	*Dollar*
vAMM	virtual Automated Market Maker	

1 Introduction

Decentralized finance, or DeFi, is an area of financial services that are provided on and operate on a blockchain. These financial services include but are not limited to derivatives trading, lending, borrowing, and earning interest trading. A derivative is a financial contract between two or more traders that derives its value from an underlying asset or group of assets (3). Since the system is decentralized, it removes intermediaries such as banks, brokerage firms and exchanges. DeFi creates a more open, fast, transparent financial ecosystem by removing intermediary parties. Anyone can open an account by setting up their cryptocurrency wallet, following which they interact with all the financial services available. Users with wallets do not need to disclose their information and can transfer their funds without permission. The software development of DeFi is usually open-source. Anyone can see the source code for the decentralized financial service they are using. This creates a transparent and authentic system. All the properties mentioned above make DeFi an essential system that can enhance how existing traditional financial systems operate.

Derivative trading has grown rapidly in the DeFi ecosphere and is one of the most popular decentralized finance services, with the total liquidity locked in derivatives trading being over $1.8 billion (4). Since its inception, derivatives trading has grown rapidly in the DeFi ecosphere. This rapid growth has led to the creation of innovative solutions to tackle various past and existing problems in the world of DeFi.

1.1 Motivation

DeFi has a lot of liquidity in derivative trading, especially in perpetual trading. The total trading volume in decentralized exchanges or DEX's exceeds $4 Billion daily (5). The algorithms that currently facilitate the trading of perpetual contracts are growing rapidly but are not perfect. The most optimal solution on the DeFi ecosystem for trading perpetuals on DEX's are automated market makers (AMMs) and virtual AMMs (vAMMs). virtual AMM's are a relatively new solution and were developed to address the problems regarding the lack of liquidity in decentralized exchanges. vAMMs were created by the Perpetual protocol team and were only deployed on the ethereum mainnet in 2020 (6).

Even though AMMs and vAMMs are highly innovative solutions, they are not the most optimal mechanisms to facilitate trading and face various issues on several frontiers. Cryptocurrencies are highly volatile securities and are prone to market volatility. Existing AMM and vAMM solutions for derivative exchanges are rigid and inflexible. AMM and vAMM based exchanges cannot adapt to market volatility since they do not have any inbuilt mechanisms to do so. Not being able to adapt its token prices with external markets for a derivative exchange causes it to depend heavily on its users to align its token prices with external mark prices. Most derivative exchanges do not have an extremely high trading volume for several perpetual markets, and relying on users to align the perpetual markets with external markets may not always be reliable.

Dynamic vAMM or DvAMM is one solution that is created to allow vAMMs to adapt their token volumes to reflect external market conditions and replicate external market movement. This project aims to explore, design, develop, and evaluate a mechanism that enables a vAMM to be configured to dynamically adjust its token volumes so that it can align a derivative exchange to external market conditions.

1.2 Outline of the Project

This project aims to design and implement a mechanism that enables configuring a vAMM to dynamically adjust its parameters so that the vAMM can adapt to external market conditions. A vAMM is dynamic when configured to adapt to market conditions hence the term dynamic vAMM.

The project's objectives include designing and implementing a configurable or dynamic vAMM for a decentralized perpetual exchange. The dynamic vAMM will be designed to allow the implemented decentralized perpetual exchange to adapt to external market conditions. This project also aims to experiment, test and observe the functionality of the implemented dynamic vAMM to evaluate if the implemented mechanism is a secure, operational and feasible solution for decentralized perpetual exchanges. This project further aims to suggest possible metrics or an algorithm that could help automate the process of configuring the vAMM to market conditions.

1.3 Dissertation Structure

The dissertation is structured as follows

Chapter 2 provides a background of the technologies and financial concepts that a reader would need to comprehend to understand this dissertation. Chapter 2 explains various technologies such as Blockchain, Automated Market Makers (AMMs), virtual AMMs and

more. Chapter 2 also introduces various financial concepts such as futures contracts and perpetual swaps.

Chapter 3 details and explains the design behind the dynamic vAMM based perpetual swap exchange.

Chapter 4 discusses how the designed perpetual swap exchange utilizing a dynamic vAMM was implemented. Chapter 4 details the implementation of core components of the implemented system and the challenges faced during their development.

Chapter 5 details the evaluation process for the implemented system. Chapter 5 explains the methodology behind evaluating the dynamic vAMM based perpetual swap exchange. Chapter 5 then provides the results of the evaluation performed and the discussion of these results.

Chapter 6 synthesizes the research and discusses the project outcomes. Chapter 6 further discusses technologies and concepts related to the current project that future projects should investigate.

2 State of the Art

This chapter covers the key concepts, techniques, and algorithms necessary to understand this research project. The concepts and technologies discussed in this chapter are important as they help understand the design, implementation and evaluation of the system being explored. Section 2.1 discusses the blockchain technology and the Ethereum blockading. Section 2.2 describes futures contracts and perpetual swaps and explains how they work. Section 2.3 discusses decentralized exchanges and details the most popular decentralized exchanges. Section 2.4 explains the working and core components of existing automated market-making protocols.

2.1 Blockchain and Ethereum

2.1.1 Blockchain

A blockchain is an expanding list of cryptographically signed, irrevocable transactional records shared by all participants in a network. Each record contains a timestamp and reference links to previous transactions. (7)

The blockchain in itself is a data structure that stores transactions. It is similar to a linked list in that the data is split into containers - the blocks. Each block is connected with its predecessor with a cryptographically secured reference. This makes the data structure tamper-evident, changes to old blocks are easy to detect and dismissed(8).

The blockchain is used to store transactions. When a transaction experiences a state change, the transaction is updated and this new transaction is stored on the blockchain.

The Bitcoin blockchain, tracks the state of units of bitcoin and their ownership. You can think of Bitcoin as a distributed consensus state machine, where transactions cause a global state transition, altering the ownership of coins. The state transitions are constrained by the rules of consensus, allowing all participants to (eventually) converge on a common (consensus) state of the system, after several blocks are mined (9).

The blockchain only supports 'add' and 'read 'operations. Therefore the previous

transactions cannot be updated or deleted. This blockchain property ensures that the information stored in the blockchain is not tampered with.

2.1.2 The Ethereum Blockchain

Ethereum is a general purpose blockchain. Similar to bitcoin, ethereum is also a distributed state machine. But instead of tracking only the state of currency ownership, Ethereum tracks the state transitions of a general-purpose data store, i.e., a data-store that can hold any data which is expressible as a key–value tuple. (9)

The Ethereum Virtual Machine or EVM The EVM is the part of Ethereum that handles smart contract deployment and execution. At a high level, the EVM running on the Ethereum blockchain can be thought of as a global decentralized computer containing millions of executable objects, each with its own permanent data store (10).

2.1.3 Smart Contracts

Smart Contracts refer to immutable computer programs that run deterministically in the context of an Ethereum Virtual Machine as part of the Ethereum network protocol—i.e., on the decentralized Ethereum world computer (11).

2.1.4 Ethereum Tokens

Ethereum blockchain tokens, also known as cryptographic tokens and cryptocurrency tokens, are transferable digital assets that are built on top of the blockchain (12). Ethereum tokens can be used to represent physical currencies such as the US Dollar and even physical assets such as gold. There are set standards for ethereum tokens with the most popular standard being ERC-20. ERC-20 is a standard interface for fungible (interchangeable) tokens, like voting tokens, staking tokens or virtual currencies (13). As of Feb 1, 2022, there are 490,216 ERC-20 tokens in circulation (14)

2.2 Futures contracts and Perpetual swaps

2.2.1 Future Contracts

A futures contract is a legal agreement to buy or sell a particular commodity asset, or security at a predetermined price at a specified time in the future (15). A futures contract is a financial derivative as it derives its value from an underlying asset or assets. A future contract allows traders to speculate whether the price of an underlying asset will rise or fall in the future. An example of an underlying asset could be gold, steel, oil, or food grains. Each futures contract has an expiration date associated with it.

Two parties are involved in a futures trade, namely, a buyer and a seller. The trader that buys a future contract undertakes the 'long' position, and the seller undertakes the 'short' position. The buyer of the futures contract agrees to buy and acquire the commodity asset from the seller. In contrast, the seller agrees to sell and deliver the commodity asset at the date when the futures contract expires. The trader that buys a future contract undertakes the 'short' position.

A futures contract transaction is carried out in a futures exchange. A futures exchange is a marketplace where a diverse range of commodities futures, index futures, and options on futures contracts are bought and sold (16). An exchange-traded futures contract specifies the quality, quantity, physical delivery time and location for the given product. (17) A sample future contract with specifications can be seen in Figure 2.1.

Important terms and processes related to futures contracts.

1. Leverage: Leverage is the ability to control a large contract value with a relatively small amount of capital. In the futures market, that capital is called performance bond, or initial margin, and is typically 3-12% of a contract's notional or cash value. (18) For example, a steel futures contract has a value of $100,000. A trader can open a futures contract and undertake a position by depositing an initial margin of $5,000 (assuming initial margin = 5% of overall contract value). Therefore, a trader has a 20X leverage on the steel futures contract ($5000 X 20 = $100,000). By investing a notional amount worth 5 percent of the overall contract price, a trader can now realise profits and losses 20 times worth their initial margin. If the leverage is higher, the risk associated with a contract increases.

2. Initial Margin and Maintenance Margin: Initial margin or a trade collateral is the amount of funds required by an exchange to initiate a futures position (19) Maintenance margin is the minimum amount that must be maintained at any given time in your account. (19). The initial and maintenance margin is set by the exchange facilitating the trading of futures contracts. Margin is a form of security to ensure that profits and losses are realised and correctly exchanged between traders. Margins are not a down payment or a sign that the trader owns the underlying commodity.

   During each trading day, the price of a commodity rises or falls. When the commodity price rises, the buyer benefits, and the seller experiences a loss and vice versa. To make sure these profits are realised, profits and losses are transferred between the buyer or seller at the end of each trading day. The profits or losses are added or subtracted from their respective accounts. This process is also known as the "daily settlement" or "Marked to market". When the amount of funds in a traders account falls below the maintenance margin, the trader either receives a margin call or is liquidated.

Corn Futures

Contract Size	5,000 bushels (~ 127 Metric Tons)	
Deliverable Grade	#2 Yellow at contract Price, #1 Yellow at a 1.5 cent/bushel premium #3 Yellow at a 1.5 cent/bushel discount	
Pricing Unit	Cents per bushel	
Tick Size (minimum fluctuation)	1/4 of one cent per bushel ($12.50 per contract)	
Contract Months/Symbols	March (H), May (K), July (N), September (U) & December (Z)	
Trading Hours	CME Globex (Electronic Platform)	Sunday – Friday, 7:00 p.m. – 7:45 a.m. CT and Monday – Friday, 8:30 a.m. – 1:15 p.m. CT
	Open Outcry (Trading Floor)	Monday – Friday, 8:30 a.m. – 1:15 p.m. CT
Daily Price Limit	$0.40 per bushel expandable to $0.60 when the market closes at limit bid or limit offer. There shall be no price limits on the current month contract on or after the second business day preceding the first day of the delivery month.	
Settlement Procedure	Daily Grains Settlement Procedure (PDF) Final Corn Settlement Procedure (PDF)	
Last Trade Date	The business day prior to the 15th calendar day of the contract month.	
Last Delivery Date	Second business day following the last trading day of the delivery month.	
Product Ticker Symbols	CME Globex (Electronic Platform)	ZC C=Clearing
	Open Outcry (Trading Floor)	C
Exchange Rule	These contracts are listed with, and subject to, the rules and regulations of CBOT.	

Figure 2.1: A sample future contract showing a futures contract specifications

(a) Margin call: A margin call refers specifically to a broker's demand that an investor deposits additional money or securities into the account so that it is brought up to the minimum value, known as the maintenance margin. (20)

3. Liquidation: Liquidation is the process of forced closing of a futures contract position. Liquidation occurs when a traders amount of funds fall below the maintenance margin, and they cannot add funds to their account to raise their balance above the maintenance margin.

4. Position: A party trading futures contract can undertake one of two positions when agreeing to open a futures contract. The positions are long and short. When a trader or organization takes a long position on a futures contract, they agree to buy the

underlying asset when the futures contract expires. Similarly, when a trader or an organization undertakes a short position of a futures trade or contract, they agree to deliver the specified quantity of the underlying asset that the futures contract is speculating on. Undertaking long and short positions can be demonstrated by the following example. Consider that a steel futures contract has a value of $100,000 for 10 1kg bars of steel. The trader that undertakes the long position would supply $100,000 and the trader that undertakes the short position would supply the 10 1kg bars of steel.

2.2.2 Perpetual Contracts

Perpetuals are the decentralized finance world's adoption of futures contracts. Perpetual contracts allows a trader or party to speculate on the price of an asset and are like futures contracts. For example, a perpetual contract can be used to speculate on the price of Ethereum.

The specifications or features of a perpetual contract can be seen in figure 2.3. The features of Perpetual Contracts are as follows:

1. Expiry of perpetuals: Perpetuals contracts have no expiration date or settlement date. This means that a trader can hold a perpetual contract for an indefinite amount of time. This feature of Perpetual contracts makes them more flexible than futures.
2. Settlement: A perpetual contract has a fixed settlement option and can only be settled using a cryptocurrency. The settlement option is usually the same token or cryptocurrency used as the margin. An example of the settlement currency can be USDT which is a blockchain-based cryptocurrency whose tokens in circulation are backed by an equivalent amount of U.S. dollars, making it a stablecoin with a price pegged to USD $1.00 (21). In contrast, futures contracts can be settled using two options. The first option is the underlying commodity asset and the second option is the cash equivalent of the underlying asset..
3. The underlying asset being speculated on is priced relative to the second asset in perpetual contracts. Typically, a volatile cryptocurrency such as bitcoin is priced against a more stable cryptocurrency or a 'stable coin'. A stablecoin is a digital currency that is pegged to a "stable" reserve asset like the U.S. dollar or gold (22). Perpetual contracts allow traders to trade on the direction of an asset without actually purchasing the underlying asset. Suppose more traders open a long position for an underlying asset 'ETH', the price of the asset 'ETH' on the perpetual market rises relative to USDC. Similarly, if more traders open a short position on an underlying asset 'ETH', the asset 'ETH' price on the perpetual market falls relative to the USD coin. The price of the asset 'ETH' relative to USDC on the perpetual market is called

the mark price of that token pair. Trading on perpetual markets typically reflects the market sentiment of the trader, and the price of the perpetual market can deviate from the spot price of an asset (see 2.2.3). Figure 2.2 shows the mark price movement concerning the number of long and short trades on a perpetual market for the ETH/USDC token pair.

Figure 2.2: Mark price movement on a perpetual market for ETH/USDC

4. Contract Specifications: Unlike futures contracts, perpetual contracts have no set specifications. Decentralised exchanges facilitating perpetual contract trading allow traders to:

   (a) Set the quantity of the underlying asset that the trader is speculating on.

   (b) Specify the leverage up to a certain limit.

5. Funding rate: As there is no delivery date in perpetual contracts, a funding mechanism is used to ensure that the price of the underlying asset in a perpetual contract is anchored to the spot market. The spot market is where cryptocurrencies are traded immediately between the buyer and seller at the current price. Funding rates are periodic payments either to long or short traders based on the difference between perpetual contract markets and spot prices. When the market is bullish, the funding rate is positive (23), and long traders pay short traders. When the market is bearish, the funding rate is negative (24) and short traders pay long traders

Feature	Description
Position	Long/Short
Collateral	The collateral or margin value
Leverage	A trader can speculate on a higher value of the underlying asset by increasing their leverage. The profits and losses will also be scaled up based on the leverage.
Entry/Mark Price	The current market price of the asset when the contract was initialized
Underlying Asset	The asset being speculated on
Amount of Underlying Asset	The amount of underlying asset being speculated on
Funding Rate	Periodic interest charged to either long or short position holders based on the difference between spot and mark prices

Figure 2.3: Perpetual Contract Features or Specifications

2.2.3 Additional Terminology

This section provides some additional terminology which is commonly used while trading futures and perpetuals.

1. Spot Market: A spot market is a market where assets are traded or exchanged instantaneously. For example, if 1 ETH is worth 23,250 USD. Trader 'A' can can buy 1 ETH from Trader 'B' instantaneously by exchanging 23,250 USD for it.

2. Spot Price: In the example above 1 ETH is worth 23,250 USD in the spot market. Therefore the spot price of 1 ETH = 23,250 USD . The formal definition of spot price from investopedia is as follows: "The current price in the marketplace at which a given asset can be exchanged for immediate delivery. Spot Prices are fairly uniform worldwide at exchanges when accounting for exchange rate(25)."

3. Mark Price: Mark price is a commonly used term in decentralized perpetual swap exchanges. Mark price describes the price of the underlying asset (usually more volatile) relative to another asset (usually more stable) in the perpetual exchange. Mark prices are local and unique to each perpetual swap exchange. For example, spot price of ETH/USDC is 1 ETH = 150 USDC and the mark price for the ETH/USDC perpetual contract is 1 ETH = 100 USD. This implies that if a trader wants to open a long position, they can deposit 100 USD with 1X leverage to speculate the price of ETH rising in the future. Suppose after a few months the mark price is 1 ETH = 140 USD, the trader can close their long position and receive a return of 140 USDC including 40 USDC of profit.

2.3 Decentralized exchanges DEXs

A decentralized exchange (or DEX) is a peer-to-peer marketplace where transactions occur directly between crypto traders. DEXs fulfill one of crypto's core possibilities: fostering financial transactions that aren't officiated by banks, brokers, or any other intermediary. Many popular DEXs, like Uniswap and Sushiwap, run on the Ethereum blockchain (26).

A DEX is typically a combination of an application layer and a smart contract-based protocol layer deployed on a blockchain. The application layer connects to the protocol layer and leverages the functionality of the smart contracts that form the protocol layer. The smart contract code and application layer code for a DEX is usually open-source and visible to everyone. For example, Uniswap is the biggest DEX in operation and code for its open smart contracts can be found on the following GitHub link: `https://github.com/Uniswap/v3-core`.

A DEX is not regulated by any authority since there is no central authority that controls the operations of a DEX. DEXs are autonomous. Since there is no central authority to regulate a DEX, the exchange rates are calculated using a smart-contract code.

There are 2 types of DEX models used to implement a DEX. The first model is the traditional order book model used by centralized exchanges, and the second model is an automated market maker or AMM model. An order book model uses an order book which is a list of all the open buy/long and sell/short orders for an asset typically organized by price (27). An order book model uses a matching system to match buy/long requests with sell/short requests for the traders on the exchange. A DEX implementing an order book model maintains the order book on the blockchain. Since the order book is stored on the blockchain, everyone can view it. When a new buy/long trade request is made with a specific amount, an order book model will try to match the buy/long request with a sell/short request with the same amounts. If a match is found, a trade is opened. If a match is not found, the matching system will store the request in the order book and execute the trade when a matching request is found. The order book model heavily relies on the volume of trades on an exchange. If the volume of trades on order book-based exchange is low, some trades may never get executed.

The second type of model is an automated market maker or AMM model. AMMs try to solve the problems faced by order book models. In centralized exchanges, a market maker participates in a market to provide liquidity to markets and open trades against user trades. In DEXs, an AMM plays the role of a market maker. An AMM exchanges tokens with any trader willing to participate in an exchange. AMMs are implemented using smart contracts on the blockchain and are composed of several mechanisms to perform their role

Decentralized exchanges have been making progress over the years but still face issues regarding liquidity. Various upcoming or existing DEXs face the issue of not having enough liquidity to facilitate accurate and seamless trading due to low trading volumes. A virtual AMM model-based exchange is a solution to deal with issues of liquidity on DEXs and is discussed in detail in section 2.4.

2.4 Automated Market Makers & virtual Automated Market Makers

2.4.1 Automated Market Makers

Automated Market Makers are a type of decentralized exchange protocol that rely on a mathematical formula to price assets (28). The function of an AMM is to facilitate trading on a decentralized exchange.

AMM's were first introduced in 2017 by Vitalik Buterin as a solution for the lack of liquidity between centralized and decentralized exchanges. (29). The traditional order book models implemented by centralized exchanges were not working well with decentralized exchanges due to the lack of liquidity. Since their introduction, AMM's have been widely adopted and are used by the most popular decentralized exchanges such as uniswap and pancakeswap (30).

The role of AMM's in DEX's is the same as the role of Market Makers in centralized exchanges or CEX's. In a CEX, market makers provide liquidity for centralized exchanges. Market makers are usually large financial organizations that provide liquidity to markets. Having liquid markets would mean that the market has a high volume of trades. For example, if a seller wants to sell an asset in an illiquid market, there may be no one to buy that asset. With the addition of market makers, a market maker may be willing to buy the asset, and therefore they help create a market. Similarly, the role of an AMM is to provide liquidity to DEX's to facilitate seamless trading.

Figure 2.4 shows the implementation of an AMM model by UniSwap Version 2.

There are two types of stakeholders that interact with AMM's:

1. Liquidity Providers or LP's that provide liquidity to an AMM, and,
2. Traders that interact with an AMM to exchange one token for another (Please refer to figure 3.1).

To provide liquidity to a DEX, AMM's require token or coin owners to provide their tokens or coins as liquidity. *The entities that provide their assets as liquidity are known as* ***liquidity***

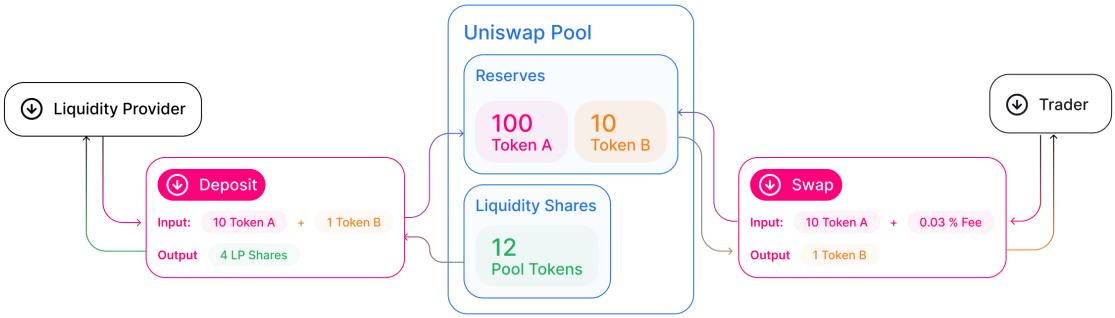

Figure 2.4: AMM model implemented by UniSwap (1)

providers. In return for depositing their assets, liquidity providers or LP's return a liquidity pool token. This process can be observed in Figure 3.1. The liquidity pool token represents the number of assets an LP has deposited in a liquidity pool. LP's earn passive income through the liquidity pool tokens that they hold.

A DEX earns passive revenue in the form of trading fees. This trading fee is paid by traders each time they initiate a trade on the DEX. The total passive revenue earned by a DEX is distributed among all the LP's. The percentage of passive revenue received by an LP depends on the number of assets deposited by an LP.

When an LP provides assets to a liquidity pool, these tokens or coins provided are added and stored in the liquidity pool. Liquidity pools are pools of tokens locked in smart contracts that provide liquidity in decentralized exchanges in an attempt to attenuate the problems caused by the illiquidity of traditional systems (31). Liquidity pools are a crucial component of AMM's as they provide an AMM with funds required to facilitate trades. AMM's also use liquidity pools to calculate the price at which a token is trading on the DEX. Liquidity pools commonly contain a combination of a volatile coin such as bitcoin and a stable-coin such as USDT. Liquidity pools can also contain stable token pairs where both the tokens stored in the liquidity pools are stable-coins. Based on the volume of assets in a liquidity pool, an AMM implements a mathematical function to calculate the price of an underlying asset.

Another important component to an AMM model is Arbitrageurs. Arbitrageurs are a type of investor who attempts to profit from market inefficiencies (32). Arbitrageurs exploit price differences for an asset between spot markets and the AMM to make profits. For example, if 1 ETH is selling for 10 USDT on an AMM and the current spot market price for 1 ETH is 11 USDT, an arbitrageur would aim to make a profit of 1 USDT by buying 1 ETH by interacting with the AMM and then selling the ETH for 11 USDT on a spot market. Arbitrageurs are core components of an AMM since they help align the price of an underlying asset to the spot market by buying and selling for risk-free profits.

In summary, AMM's allows traders to buy and sell an underlying cryptocurrency using an algorithm. This algorithm regulates and sets the asset's price based on how much of this asset is available. As more traders hold a 'long' position on one cryptocurrency, buying the cryptocurrency becomes more expensive as the supply decreases and demand increases. If traders short or sell the underlying cryptocurrency more, the cryptocurrency gets cheaper because there is a higher supply and lower demand

2.4.1.1 Constant Function Market Makers or CFMM's

CFMM's are a type of automated market maker that are defined by their trading function and its liquidity pool reserves (33). CFMM's ensure that there will always be liquidity present in a liquidity pool as the trading functions used have a fixed bonding curve for asset price determination. This curve changes depending on the ratio of assets in a liquidity pool. There a three main trading functions that are commonly used. These trading functions are

1. Constant Product Market Makers: These market makers operate on liquidity pools with only 2 assets and not more. The relationship between the two assets is non linear. The trading function for a constant product MM is:

$$assetA * assetB = constant$$

   Figure 2.5 shows the relationship between asset volumes and the constant k for the Constant Product Market Maker.

2. Constant Sum Market Maker: These market makers operate on liquidity pools with only 2 assets and not more. The relationship between the two assets is linear. The trading function for a constant sum MM is:

$$\sum_{i=1}^{n} R_i = k$$

   R_i represent the reserves of the each asset in the liquidity pool and k represents a constant value.

   Figure 2.6 shows the relationship between asset volumes and the constant k for the Constant Sum Market Maker.

3. Constant Mean Market Maker: A constant mean market maker is a generalization of a constant product market maker, allowing for more than two assets. (34). Constant mean MM's enable the addition of weights which allows for the addition of two or

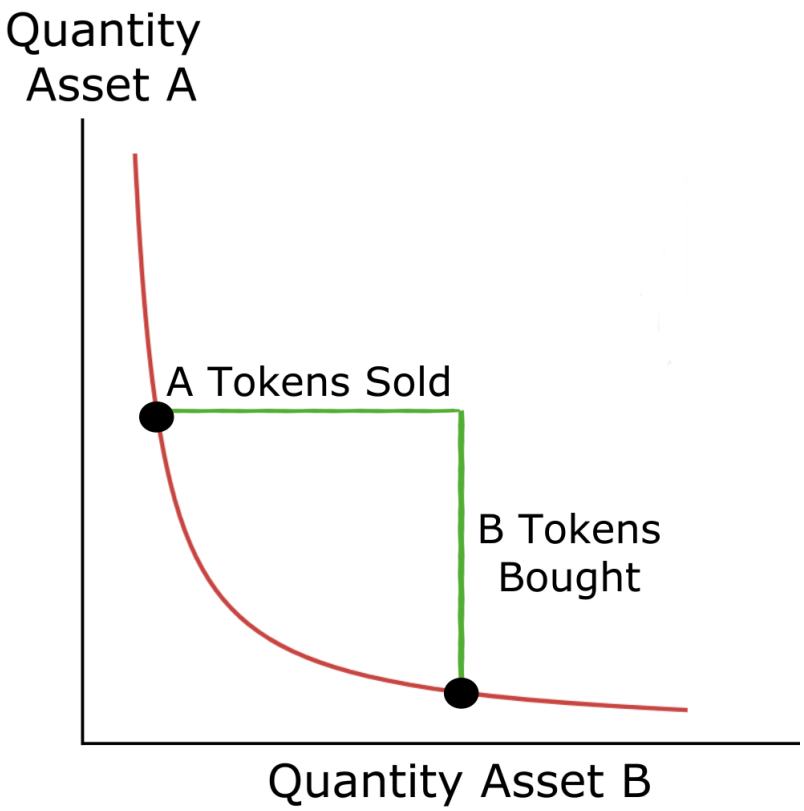

Figure 2.5: Graph of constant product market maker (2)

more assets. The trading function for a constant mean MM is:

$$\prod_{i=1}^{n} R_i^{w_i} = k$$

R_i represent the reserves of each assets in the liquidity pool. W_i represents the weight of each asset and k represents a constant value (34).

Problems with AMM's:

- Impermanent Loss: Impermanent loss happens when you provide liquidity to a liquidity pool, and the price of your deposited assets changes compared to when you deposited them. The bigger this change is, the more you are exposed to impermanent loss. In this case, the loss means less dollar value at the time of withdrawal than at the time of deposit (35).
- Slippage: Slippage occurs when a trade's size negatively affects a trader's rate of return. AMM's experience slippage when large trades are initiated or if the market is volatile. This is so because AMM's typically use CFMM's with a constant value k. If the constant k is a low value and a large buy order for an asset is placed on an AMM, the volumes of the asset in the liquidity pool will be largely affected by the k value, and the AMM will price this asset at a high value since its supply is diminishing. The

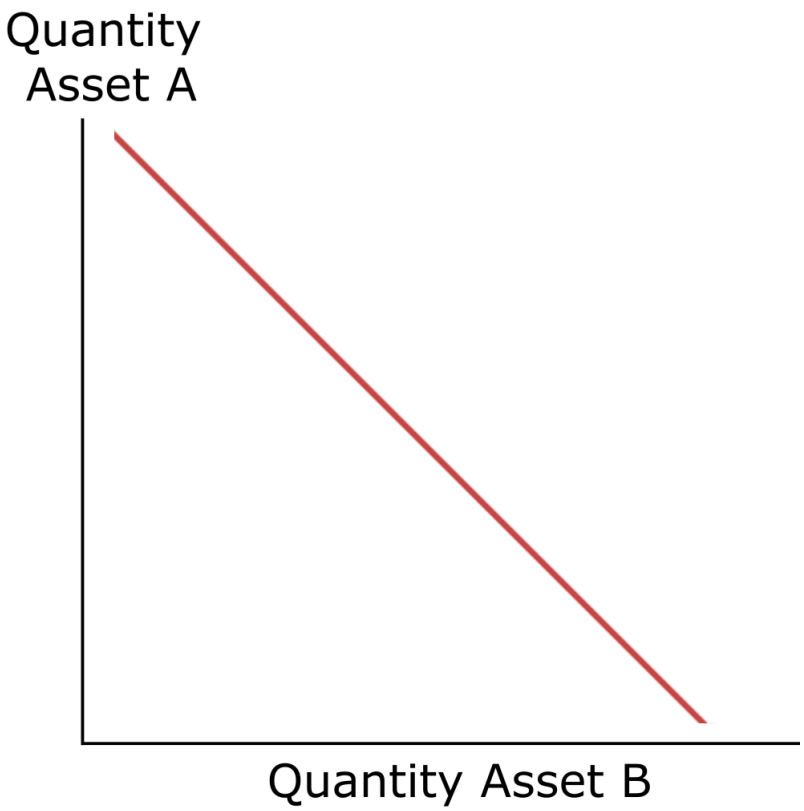

Figure 2.6: Graph of constant sum market maker (2)

trader may experience a higher price than spot markets and may experience a loss.

- Lack of shared liquidity model: Each AMM has their own dedicated liquidity pool because of which AMMs cannot share liquidity. If AMMs want to share liquidity, it would result in large design complexities.

2.4.2 Virtual Automated Market Makers

The concept of virtual AMMs is derived from the mechanisms of AMMs. Virtual AMMs were created to deal with issues present with AMM, such as Impermanent loss. Rather than having the concept of liquidity providers and liquidity pools, vAMMs use virtualized tokens and remove the need for liquidity pools. The VAMM is only used for the asset price calculation and how this is affected once a trade is executed.

Mechanism of vAMM and how a trade is performed: In VAMM's, whenever a user wants to take a long or short position on an asset, they place their position, the VAMM calculates the price accordingly, and the user's margin is stored in the vault as collateral. Since the traders are providing the funds for trading, we do not need liquidity providers, which solves the problem of impermanent loss since there are no liquidity pools. The traders themselves provide the funds. The higher the number of traders, the larger is the vault size and liquidity contained by the DEX. The difference between the implementation of AMMs

and vAMMs can be seen in figure 2.7. A vAMM does not need a liquidity pool, and the vault can match any funding issues present. (36)

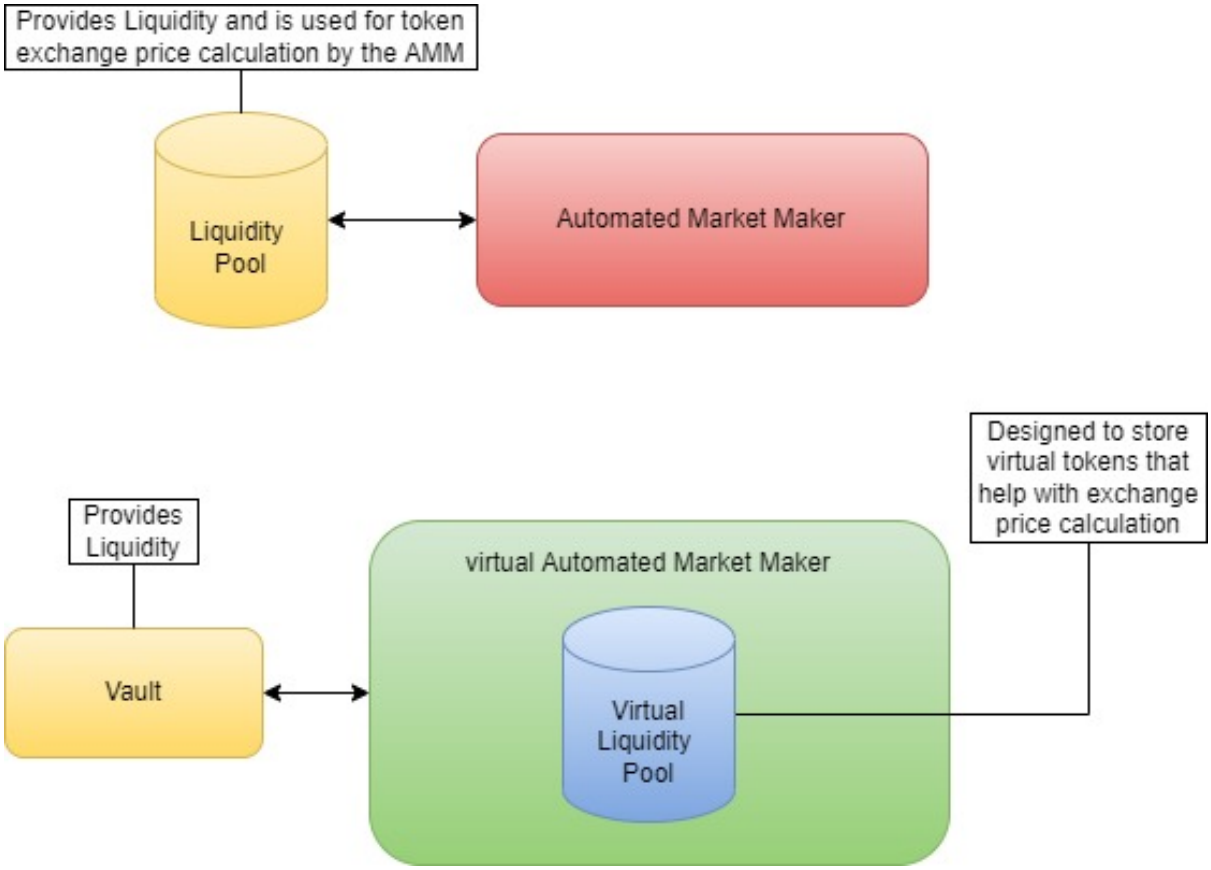

Figure 2.7: Difference between AMMs and vAMMs - Liquidity Pools and Vault

Differences between real and virtual liquidity pools:

1. Virtual liquidity pools do not store or consist of any real tokens. Virtual liquidity pools store virtual tokens to represent the value of the real tokens. For example, in a virtual liquidity pool, the reserves of the token ETH are represented as vETH. Virtual liquidity pools are only used to calculate the exchange prices of tokens; they do not provide liquidity.
   Real liquidity pools store real tokens. The tokens stored in a real liquidity pool are exchanged for other tokens.

2. From figure 2.7 we can observe that a virtual liquidity pool is contained inside the vAMM, whereas a real liquidity pool is stored outside an AMM. A DEX implementing an AMM allows users to directly interact with the liquidity pool. A DEX implementing vAMMs is only used for token price calculation and does not allow users to interact with the virtual liquidity pool.

Example of a vAMM in operation: Let us consider an example of a vAMM that is used to exchange ETH and USD coin (USDC). The vAMM implements a constant product market

making function and the formula can be seen in equation 1. vETH represent the number of virtual ETH tokens and vUSDC represent the amount of virtual USDC tokens.

$$vETH * vUSDC = k \tag{1}$$

Initial state of the vamm:

- vETH = 10
- vUSDC = 1000
- k = vETH * vUSDC = 10,000
- Mark price: 1 ETH = 100 USDC

1. Trader 'A' requests to open a long position on ETH with 10 USDC as collateral with 10X leverage.

   (a) Total contract value: 10USDC X 10(leverage) = 100vUSDC

   The collateral of 10 USDC is stored in the vault and since the trader 'A' has gone long, the reserves of vUSDC are increased by 100. vUSDC increases to 1100 and to calculate the new vETH reserves, we perform the calculations seen in equation 2

$$new\ vETH = \frac{10,000(k)}{1100(new\ vUSDC)} = 9.09 \tag{2}$$

$$vETH\ allocated\ to\ trader\ A = new\ vETH - old\ vETH = 10 - 9.09 = 0.91 \tag{3}$$

   On opening the long trade, trader 'A' gets allocated with 0.91 vETH at an entry price of 100 USDC per ETH. The new mark price of ETH/USDC is = 121 USDC per ETH. The mark price is calculated by dividing the vETH reserves with the vUSDC resereves.

2. After a month of requests, the new virtual liquidity pool reserves are:

   - vETH = 8.33
   - vUSDC = 1200
   - k = 10,000
   - mark price 1 ETH = 144 USDC
   - vault balance = 20 USDC

   Trader 'A' now wishes top close the trade that was opened with 10 USDC collateral with 100 USDC/ETH as entry price. The 0.91 vETH allocated to trader 'A' are added

back to the virtual liquidity pool and we find the new vUSDC reserves and the total amount that needs to be returned to trader 'A'. The caluclations can be seen below:

$$new\ vUSDC = frac10,000(k)8.33 + 0.91(new\ vETH) = 1082.25 \tag{4}$$

$$total\ contract\ return\ value = old\ vUSDC - new\ vUSDC = 1200 - 1082.25 = 0.91 = 117.75 \tag{5}$$

$$total\ amount\ returned\ to\ trader\ A = \frac{total\ contract\ return\ value}{leverage} = \frac{117.75}{10} = 11.775 \tag{6}$$

From the calculations above, we can see that trader 'A"s return value after closing the contract is 11.775 USDC. This amount is withdrawn from the vault and returned to trader 'A'. Since trader 'A' closed their long position when the mark price was higher than the entry price for the trade, trader 'A' earned a profit of 11.775 - 10 USDC = 1.775 USDC.

Issues with a vAMM's:

1. vAMM's are a good solution to the issues faced by AMM's but are not the most efficient solution. vAMM's use a trading function to price assets, and the trading function can be optimised to be at the most liquid zone of the trading function curve. However, vAMM's do not allow for the adjustment of the constant in the trading function and cannot optimise the trading function. The reason for this is the lack of funds. Since traders deposit their own funds, there is no flexibility available in the vault of a vAMM to adjust the trading functions.
2. vAMM's rely on arbitragers and funding payments to bring the asset price on the DEX equal to the spot price. If a DEX does not have a high trading volume there will be fewer arbitragers. This issue can be tackled by dynamically adjusting the value of an asset on a DEX to be the same as the spot price.

   Dynamic virtual automated makers or DvAMM's solve the problem of vAMM's by optimizing the trading function. This is done by the dynamic adjustment of the constant value in the trading function. DvAMM's also reduce the dependence on arbitragers by having a price readjustment mechanism so that the price of an asset on the DEX matches the spot price of that asset.

2.4.3 Dynamic Virtual Automated Market Makers

In traditional AMMs and vAMMs, there is no mechanism to adjust the mark price of a given token pair to their spot price. If the cryptocurrency markets are volatile, the spot exchange rate of one token relative to another can change drastically. AMM and vAMM based perpetual exchanges have no mechanism to adapt to market conditions and volatility. Instead, AMM and vAMM based exchanges rely on funding payments and arbitrageurs to match the mark price to the spot price for a given token pair. These mechanisms can fail when there are few arbitrageurs trading on the perpetual swap exchange or if the number of users trading on the perpetual swap exchange is low.

To make vAMM's more sensitive to external market conditions, dynamic vAMMs are implemented. Dynamic vAMMs make a vAMM configurable by providing a mechanism to update the asset reserves and the constant k of the virtual liquidity pool,

To adapt to external market conditions, a dynamic vAMM can update its token reserves and the K value to match the spot price for a particular token pair. Updating the token reserves of the dynamic vAMM changes its mark price. The following example demonstrates how a dynamic vAMM may choose to update a token pair's mark price to match the spot price.

- A constant product market marker with vETH and vUSDC tokens is implemented
- Initial token reserves are vETH = 10, vUSDC = 1000, k =10000 and mark price (vUSDC/vETH) is 1 ETH = 100 USDC
- The spot price of ETH/USDC is 1 ETH = 150 USDC
- The dynamic vAMM wants to update the mark price to match the spot price and therefore it chooses to configure its virtual token values and k to do so.
- Keeping the vUSDC tokens constatnt, the mark price can be updated by finding new vETH reserves to update the mark price. The new vETH tokens to reflect mark price = spot price can be seen in the equation below.

$$(\textit{new mark price})150\textit{USDC} = \frac{1000\,(\textit{vUSDC})}{\textit{new vETH reserves}} \tag{7}$$

- new vETH reserves = 1000/150 = 6.66666.
- On finding the new vETH reserves, the dynamic vAMM can update its K value to be equal to new-vETH-reserves * vUSDC-reserves = 6.66666*1000 = 6666.66
- The new mark price is udpated and is equal to 1000vUSDC/6.6666vETH = 150.00 USDC per ETH

Similarly, if a dynamic vAMM wants to reduce the amount of slippage experienced by a perpetual swap exchange, it can scale up its K value by scaling its virtual token reserves. One way to scale the k value for constant product market makers is to multiply all the values in the equation with a particular value.

A dynamic vAMM provides mechanisms to configure a vAMM to adapt to external market conditions. A dynamic vAMM also provides mechanisms to scale the k value to a higher value to reduce slippage (6) (37).

This research aims to design, implement, test and evaluate a system capable of adapting its mark price so that the system can adapt to external market conditions. This research implements a dynamic vAMM that can scale its virtual token reserves and k values to adapt to external market conditions to achieve these objectives.

Currently, only one protocol implements a dynamic vAMM in a production environment. This protocol is known as drift protocol (38) and was launched on October 25, 2021. Drift protocol has a highly customized way of configuring a vAMM. This research does not build on drift protocols implementation of a dynamic vAMM and tries to implement a different way of configuring a vAMM. Drift protocol was discovered halfway through this research project, and attempting to implement their method of implementing a dynamic vAMM would require more time.

3 Design

A proof-of-concept perpetual swap exchange implementing a dynamic vAMM was designed and implemented to experiment, analyse and evaluate the performance, benefits and pitfalls of dynamic virtual AMMs. The design of the dynamic vAMM based perpetual swap exchange is discussed in this chapter

3.1 Overview of the System

The system being designed is a dynamic vAMM based perpetual swap exchange. The engineered system aims to design and implement a dynamic or completely configurable vAMM. The designed system will allow the researcher to test and experiment with different configurations on a vAMM and observe the impact of such configuration changes of the exchange. The system allows the researcher to:

1. Open and Close long or short perpetual contract requests.
2. View open trades.
3. View the internal state of perpetual swap exchange. This includes viewing the details of the virtual liquidity pool, the amount of user collateral in the vault, and the total volume of USDC locked in long and short trades.
4. Manually feed-in spot prices for a token pair for the purpose of experiments. For example, the researcher can inform the perpetual swap exchange that the spot trading price of 1 ETH is 100 USDC.
5. Configure the vAMM and adjust the token reserves in the virtual liquidity pool using the user interface. This feature would allow the researcher to test different configurations on the vAMM.

The dynamic vAMM based perpetual swap exchange is currently designed to trade the ETH/USDC perpetual contract. USDC stands for USD coin, and ETH stands for the token Ether. The system being designed is structured in 2 layers: the application and the dynamic vAMM (DvAMM) protocol layers.

1. The Application Layer: The app layer consists of 2 components, the user interface (UI) and the blockchain layer interaction service (BLIS) . The user interface allows traders to execute trades and interact with the dynamic vAMM based perpetual swap exchange. BLIS is designed to communicate user requests from the UI to the DvAMM protocol layer
2. The Blockchain or DvAMM Protocol Layer: The DvAMM protocol layer is analogous to the back-end of the perpetual swap exchange. The DvAMM protocol layer consists of the smart contracts that facilitate perpetual trading on the perpetual swap exchange. The main responsibility of this layer is to provide services that facilitate perpetual swaps on the exchange and allow the researcher to test different configurations on the vAMM. The responsibilities of the blockchain or DvAMM protocol layer are as follows:
   (a) Accurately calculating token exchange prices when a trade is executed with the help of a dynamic vAMM
   (b) Securely storing each user's funds. When a user opens a trade, these funds are provided as margin or collateral.
   (c) Handling the profit and losses of each user.
   (d) Providing a mechanism that allows the researcher to configure the vAMM.
   (e) Keeping track of the total amount of liquidity locked in long positions and short positions. This would allow the perpetual swap exchange to keep track of the balance between trades.

Figure 3.1 shows the design of the DvAMM based perpetual swap exchange.

3.2 The Application Layer

The goal of the application layer is to provide a user-friendly interface that allows users to execute trades on the DEX. The application layer design consists of the User Interface (UI) and the Blockchain Layer Interaction Services. The UI is composed of a user view for users and a test view for performing experiments on the dynamic vAMM.

3.2.1 User Interface (UI)

The User Interface or UI is designed to allow the system's users to interact with the various services provided by the designed system. The UI is designed to be simple, user-friendly and descriptive so that users of all levels (novice or experts in trading) can easily understand and use the functionalities provided by the designed perpetual swap exchange.

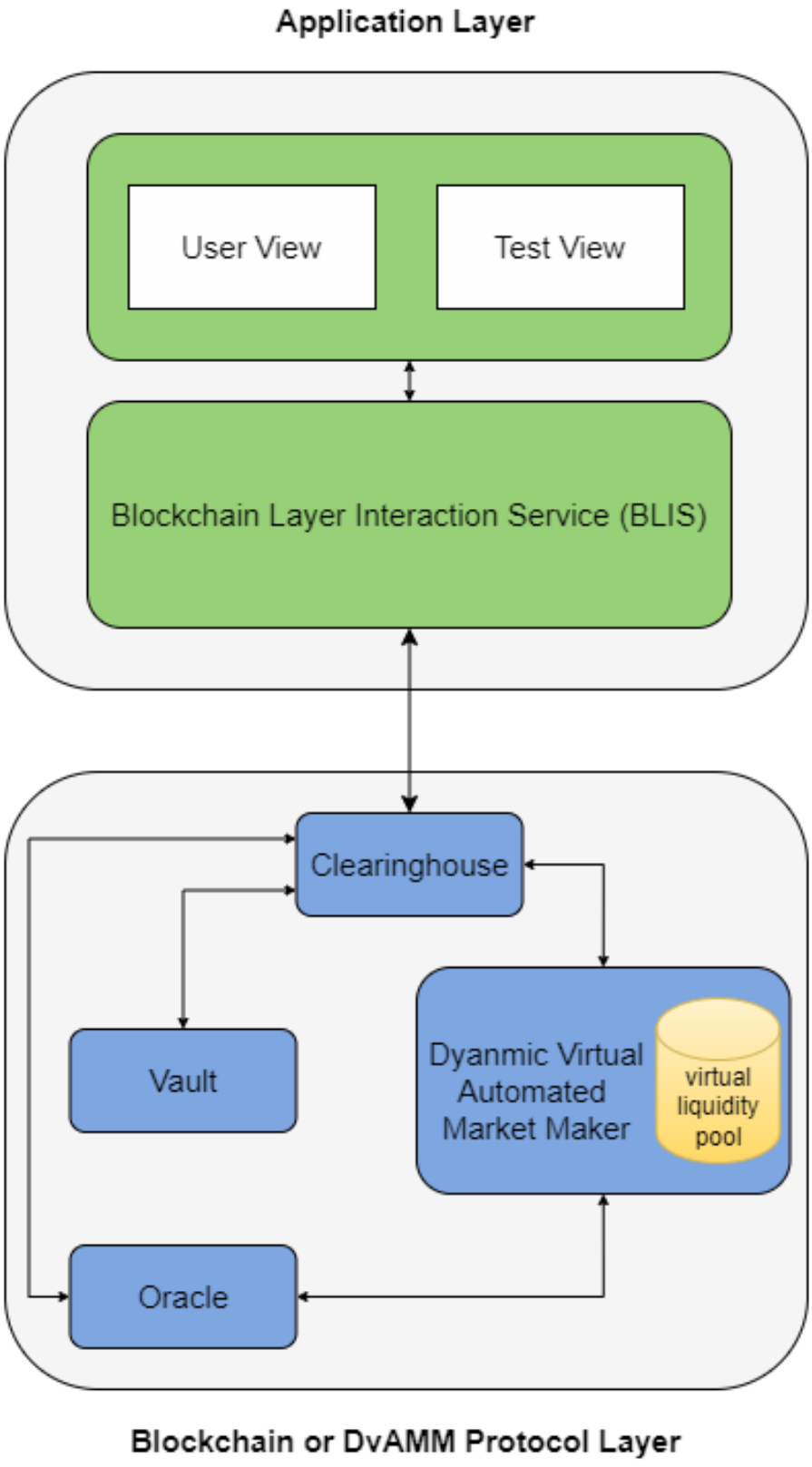

Figure 3.1: System Architecture

The UI is designed with 2 separate views for two user groups. A user view is designed for traders that want to open or close perpetual contracts on the DvAMM DEX. A test view is designed for the researcher to interact with and perform experiments on the dynamic vAMM. The test-view allows a researcher to test and experiment with the dynamic vAMM to observe the behaviour of the dynamic vAMM when its state changes.

The user view is designed to primarily allow users of the designed system to open new perpetual contracts and close existing perpetual contracts. The user view allows traders to view their current open perpetual contract positions. The user view also allows traders to connect their Ethereum wallets to the designed perpetual swap exchange using metamask. Metamask is a browser extension that allows traders to connect to their Ethereum wallets (39). Finally, the user view displays important pieces of information. A details component is designed on the UI to provide information regarding the current mark price of the token pair (ETH/USDC) being traded, the amount of funds a trader has stored on the designed exchange and the amount of funds stored in the Vault. The details component also provides data regarding the current state of the dynamic vAMM and information about the spot price of the token pair being traded on the exchange.

The test view is designed to have a few additional features and functionalities over the user view. The designed additional features allow the user to perform experiments on the dynamic vAMM by using the UI. The test view allows the researcher to configure the mark price of the ETH/USDC token pair by interacting with the designed UI. For example, if the current exchange rate of 1 ETH is 100 USDC, the researcher can update the exchange rate of 1 ETH = 150 USDC by clicking a button on the UI. Updating a token pair's mark price changes the reserves of tokens in the virtual liquidity pool and directly impacts the profits and losses of users. This feature would enable conducting experiments on the dynamic vAMM. The test view also provides a feature of a mock oracle to feed in mock spot prices of the ETH/USDC token pair. The mock oracle on the UI allows the researcher to provide the spot price of the ETH/USDC token pair to the blockchain or a DvAMM protocol layer.

The test view and user view are designed to have a faucet. A faucet is designed to provide the researcher and traders with local USDC tokens for testing. The test cryptocurrency will help the researcher and traders pay for trades when testing the perpetual swap exchange. Using a local test USDC currency provides a level of convenience for testing the functionalities of the exchange as users, and the researcher do not need to find faucets to request funds from to trade and test the designed exchange.

Figure 4.1 shows the user view UI and 4.2 shows the test view UI. The user view UI consists of the following tabs, components and features:

1. A button to connect to metamask.

2. Opening a trade tab: The open-trade tab is designed for a trader to specify the amount of USDC they want to deposit as collateral for taking a long or short position on ETH. For example, If the price of 1 ETH = 100 USDC on the perpetual exchange and Trader A wants to take a long position on ETH for 10 USDC, Trader A can input the amount as 10 USDC on the open trade tab as collateral. Trader A has now taken a long position on 0.1 ETH
3. View Open Trades tab & Close Trade: The View Open Trades tab allows traders to view their current open perpetual contract positions. The view open trades tab displays the following for each trade:
   (a) Trade position, i.e. long or short
   (b) The amount of underlying asset that a user has taken a long/short position on.
   (c) The entry price of the trade or the price of 1 ETH in USDC when a user opens a trade.
   (d) A close trade button to close a trade
4. Details Component: A details component is designed to allow traders to view their local USDC balance, the Vault's balance, the mark and spot prices of ETH/USDC and information regarding the state of the dynamic vAMM. The details component on the user view also has a deposit button that allows the user to access the faucet and request funds from

The test view is designed to have additional functionality over the user view to allow the researcher to test. The additional functionalities available in the test view are:

1. Configure DvAMM tab: This functionality is designed to allow the researcher to update the exchange rate of a token pair.
2. A Mock Oracle tab
3. A Faucet component: A faucet is designed to provide the researcher with local USDC tokens for testing.

3.2.2 Blockchain Layer Interaction Service (BLIS)

BLIS is designed for the application layer to communicate and interact with the blockchain or DvAMM protocol layer. The smart contracts containing the system logic are deployed on a blockchain, e.g. the Ethereum blockchain. BLIS uses MetaMask to connect to the blockchain that holds the smart contract code. BLIS is designed to make appropriate functions calls in the clearinghouse smart contract to perform specific actions that a user would like to carry out.

For example, if a trader clicks on the close trade button on the UI, BLIS will first formulate the user's request with the required parameters and then transmit the request to the DvAMM protocol layer. The request will call the closePosition function in the clearinghouse and pass the required parameters to the closePosition function.

BLIS is designed to hide the complexity of interacting with smart contracts away from the users of the DEX.

3.3 Blockchain or DvAMM Protocol Layer

The Blockchain or DvAMM protocol layer is designed to house the logic that provides the functionalities of a perpetual swap exchange. The goal of the DvAMM protocol layer is to process requests for opening and closing perpetual contracts. The DvAMM protocol layer is designed to handle perpetual contract requests by implementing a totally configurable vAMM and a vault. The vAMM is designed to calculate the mark exchange rate of the tokens whenever a perpetual contract is opened or closed. The vault is designed to hold the collateral or margin deposited by each user securely. For a DEX implementing a vAMM, the losses of a short position offset the profits of a long position and vice-versa. A decision was made during the design process to design and implement an in-house dynamic vAMM. This project did not build upon existing AMM and vAMM protocols because of the following reasons:

1. The low number of projects implementing a virtual liquidity pool in a production environment. There are only 2 protocols – Perpetual Protocol reference and Drift Protocol reference that implement vAMM based perpetual swap exchanges in production.
2. Incompatibility:
   (a) Drift protocol reference is the only protocol that leverages a dynamic vAMM. However, Drift Labs have written their smart contracts using Rust and have deployed their protocol and exchange on the Solana blockchain. The current project is being built on the Ethereum blockchain, and the smart contracts are being written in Solidity.
   (b) Perpetual Protocol (perp.fi) is built on the Ethereum ecosystem and implements a vAMM. However, it is not possible to locally deploy and test perpetual protocols smart contracts since they are not compatible with local deployments. Building on top of perpetual protocol would require deep knowledge of how perpetual protocol develops, deploys, and tests its protocol.
   (c) Drift and perpetual protocols had an immense amount of smart contracts in their

codebases. Understanding their entire codebase and building on top of them would be more difficult than implementing an in-house protocol.

3. To achieve this project's objectives, this project needs an environment suited to conducting a series of experiments. The perpetual swap exchange needs to provide tailored functionality that can help carry out intended experiments on the vAMM. Building on top of existing protocols would prevent experimenting with different configurations on the vAMM. It would be complex to build up upon existing vAMM and AMM protocols. Furthermore, building an in—house protocol will help understand the advantages and limitations of a dynamic vAMM.

Drift protocol and perpetual protocol served as a reference during the design and implementation process, even though this project did not directly build upon their code-bases.

To build a DvAMM protocol layer tailored to the requirements of the current project, it is important to understand the functional requirements. The purpose of the DvAMM protocol layer is to process perpetual contract requests with the help of a configurable or dynamic vAMM and a vault. The vault and dynamic vAMM will help in handling tokens and user collateral. The functional requirements for this protocol are as follows:

1. The protocol should be able to open and close perpetual contracts.
2. The protocol should provide the exchange rate for a token pair being exchanged on the perpetual swap exchange. Furthermore, the protocol should adjust the token exchange rate whenever a perpetual contract is opened or closed.
3. The protocol should securely store all user collateral deposits in a vault.
4. The protocol should have a mechanism to receive spot market exchange prices for a token pair.
5. The protocol should allow configuring the vAMM so that the perpetual swap exchange can adapt to market conditions for a given token pair.
6. The protocol should return information regarding the open positions of a user.
7. The protocol should return details about the vAMM token reserves, the total liquidity locked in the vault and the perpetual swap exchange rates for a given token pair.

4 components were designed to satisfy the functional requirements of the DvAMM protocol layer. Figure 3.2 shows a high overview of the designed operations for each component of the DvAMM protocol layer. Each of these components will be discussed in further detail in the sections below.

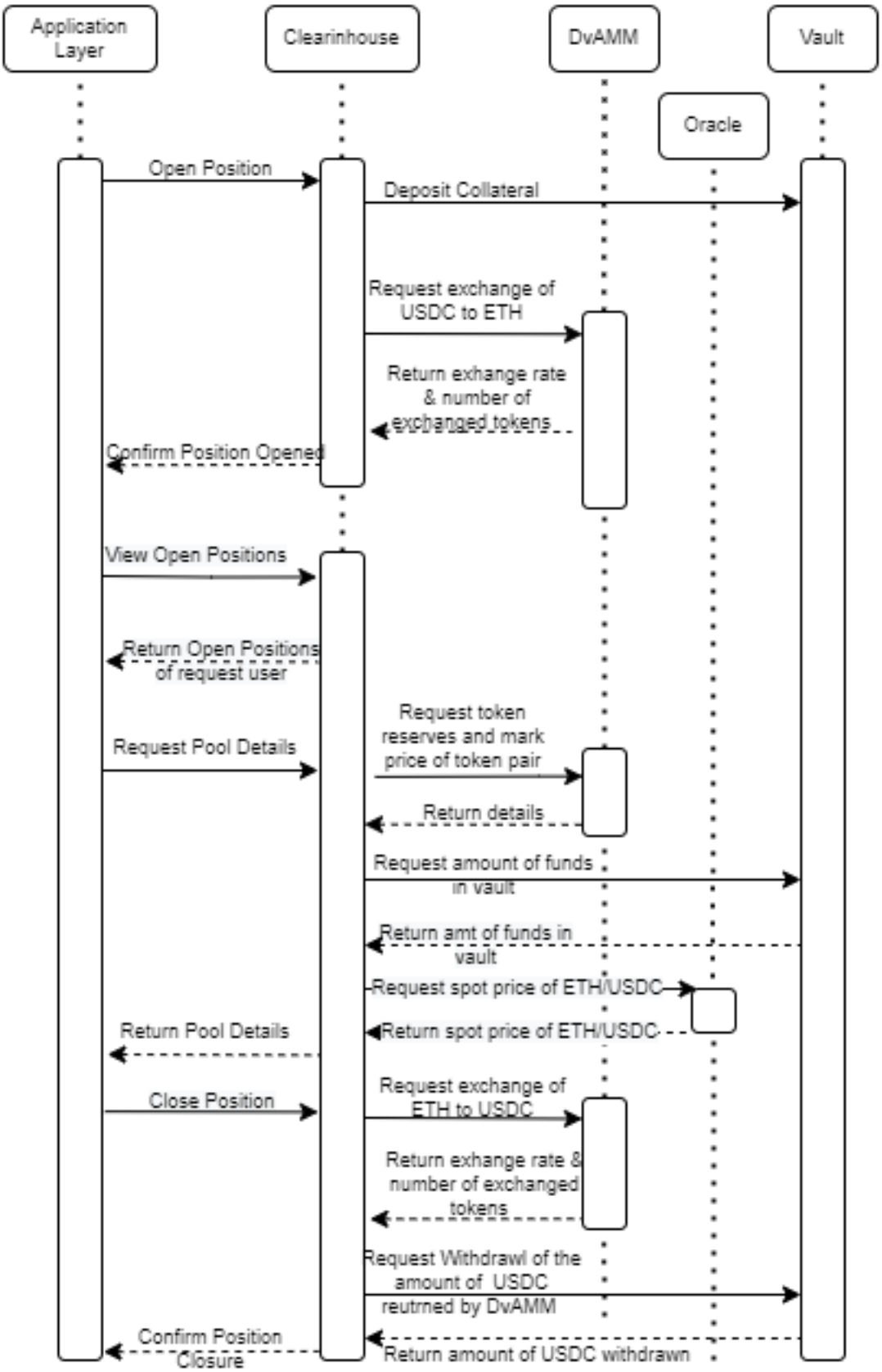

Figure 3.2: Overview of Operation of the DvAMM Protocol Layer Components

3.3.1 Clearinghouse

The clearinghouse is designed to enable user interactions with the perpetual swap exchange. The clearinghouse is responsible for initializing local user accounts on the exchange and implementing opening and closing perpetual contract requests. The clearinghouse is designed to accept user collateral payments from a local user account on the exchange and securely deposit the collateral in the vault. The collateral or margin payments are accepted in USDC.

The clearinghouse smart contract is also designed to allow users to store funds for trading on the designed exchange. Users can use funds from their local exchange accounts to provide collateral for opening a new perpetual contract. A later version of the protocol layer would aim to provide the functionality of users providing collateral directly from their browser connected blockchain wallets using ERC-20 tokens as it provides a more seamless and trustworthy trading experience. The clearinghouse is designed to securely return a user's profits and losses when they close an open position to their local perpetual exchange USDC wallet/account. The clearinghouse is also responsible for returning details about the vault, token reserves in the vAMM and the mark price of a token pair. The clearinghouse is responsible for handling funding payments in drift and perpetual protocols. However, a design choice was made not to include funding payments in the current design. Funding payments are not designed and implemented because of time constraints. Designing and implementing the core functionality of the dynamic vAMM is more critical for this study than implementing funding rates. This study aims to design and implement a dynamic vAMM to observe the effect of configuring a vAMM on the exchange as a whole. Implementing funding payments would require more time and would not contribute to achieving the research objectives.
Table 3.1 describes the external and internal functions designed to be provided by the clearinghouse and their purpose.

3.3.2 Dynamic vAMM

A dynamic vAMM is a totally configurable vAMM. The purpose of the vAMM is to provide a price discovery mechanism for the exchange rates of a token pair. In simple terms, the vAMM is designed to calculate the price of one token relative to the other.
The vAMM consists of a virtual liquidity pool that keeps track of the number of virtual tokens (for example, virtual ETH and virtual USDC). The vAMM design implements the constant product market maker (CPMM) in a trading function. A CPMM is only able to handle two tokens at a time. Therefore, the design will only be able to handle 2 tokens at a time.
The formula for a CPMM can be seen in equations 1 and 2. Equation 2 elaborates on

Table 3.1: Clearinghouse smart contract function specifications.

Function	External/Internal	Purpose
Open Position	External	Open a long or short perpetual contract for a given user. Deposit the collateral for the open position in the vault. Keep a record of the transaction
Close Position	External	Close a long or short perpetual contract for a given user. Withdraw the collateral from the vault for the trade being closed, and securely return it to the user's account. Keep a record of the transaction
Get Pool Details	External	Return the details of token reserves in the vAMM, return the amount of USDC in the vault, and return the price of 1 ETH in USDC.
Get total volume of open long and short positions	Internal. Not exposed to users	Return the total amount of USDC contributing to long trades and the total amount of USDC contributing to short trades.

equation 1 where x represents the volatile asset reserves stored in the virtual liquidity pool and similarly y represents the stable asset reserves i.e. USDC. k is a constant and is designed to be equal to the product of the stableAssetReserves and volatileAssetReserves.

$$x * y = k \tag{1}$$

$$volatileAssetReserve * stableAssetReserve = k \tag{2}$$

$$price = \frac{stableAssetReserve\,(number\ of\ USDC\ tokens)}{volatileAssetReserve\,(number\ of\ ETH\ tokens)} \tag{3}$$

In the current design, the token pair used is ETH and USDC. Therefore, the job of the vAMM is to keep discovering the price of ETH relative to USDC as perpetual contracts are opened and closed. ETH would represent the volatileAssetReserves and USDC would represent the stableAssetReserves in equation 2. The mechanism to discover the price of 1 ETH is USDC in the designed trading function can be seen in equation 3. A vAMM can be initialized for more token pairs if the system admins or the researchers decide to do so.

The perpetual swap exchange rate for a token pair changes as a trade is opened or closed. When a trade is opened, the vAMM takes in the value of the user collateral. The vAMM allocates vETH tokens to a user based on the user collateral for the trade. The allocated vETH tokens represent the amount of ETH a user has opened a trade on. When a trade is closed, the vAMM takes in the amount of vETH tokens a user had bet on. Since the trade is being closed, the vAMM adds the allocated vETH tokens back to the liquidity pool. The vAMM then returns the amount of vUSDC tokens that represent the value of the vETH

tokens added back to the pool. This process can be seen in figure 3.3

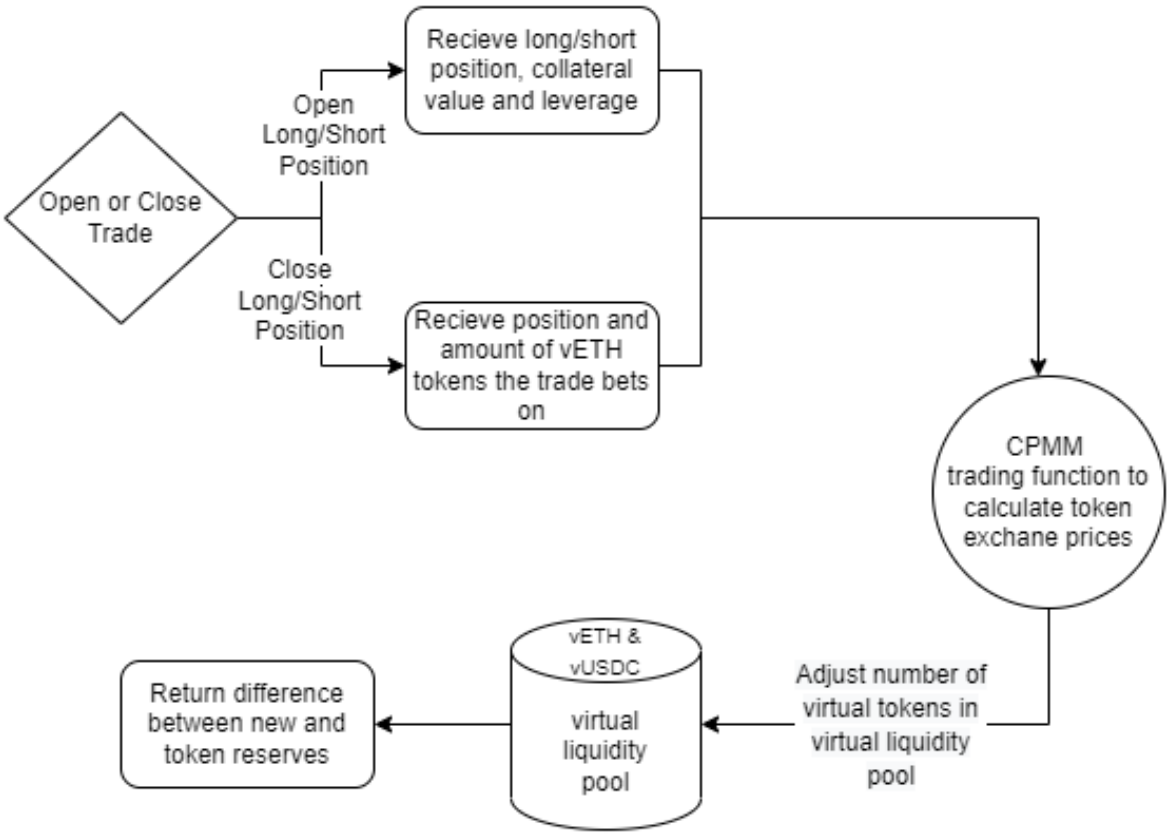

Figure 3.3: Mechanism of token exchange price calculation using a vAMM

3.3.2.1 Configurational vAMM

In traditional AMMs and vAMMs, there is no mechanism to adjust the mark price for a given token pair. The lack of flexibility in traditional AMMs and vAMMs reduce their ability to adapt to market conditions. The DvAMM protocol is designed to provide mechanisms to recalibrate liquidity and token pair mark prices based on spot prices for a token pair.

The protocol being designed aims to develop and test a more flexible vAMM. To make a vAMM more flexible, it needs to be configurable. A vAMM can be configured by scaling the value of either the volatile asset reserves or the stable asset reserves seen in equation 2. Scaling either token reserves updates the mark price of the token pair(see equation 3). After scaling the token reserves, the value of the constant k is also updated to be equal to the product of the new token reserves.

The protocol being designed scales the volatile asset reserves to change the exchange rate for a token pair. The protocol being designed provides functionality to adjust the mark price to be closer to or equal to the spot price for a given token pair by adjusting the value of the volatile asset reserve and k. This functionality is only available to the researcher so that

different vAMM configurations and their potential effects on the protocol and users can be studied. The functions designed to configure the vAMM and their descriptions are provided in table 3.2

Table 3.2: Functions designed to configure the vAMM

Function	Parameters	Purpose
Update Mark Price to Spot Price	None	Updates the mark price for a given token pair to the spot price. For example, if the mark price is 1 ETH = 100 USDC and the spot price is1 ETH = 200 USDC, calling this function will update the mark price to 1 ETH = 200 USDC
Update Mark Price on demand	Provide the new exchange rate that you want for the given token pair. For example, if you want to update the price of 1 ETH to 115 USDC. Provide 115 USDC as the parameter	Updates the mark price for a given token pair to the amount of USDC provided as a parameter.

3.3.3 Vault

The smart contract vault is designed to hold the collateral deposits securely for the open perpetual contracts for all users. The design for the vAMM and vaults are drawn from perpetual protocols design. For a vAMM based exchange, traders provide liquidity to each other by depositing collateral (40). This collateral needs to be stored by the smart contract vault The vault is designed to accept user collateral deposits when a new perpetual contract is opened. The vault is also designed to return a user's profits and losses when a user closes an open perpetual contract. The vAMM and vault smart contract are also designed to ensure that there will never be a situation where the vault will not have enough collateral to pay back all the traders trading against a vAMM (36)

3.3.4 Oracle

The DvAMM protocol needs to have a mechanism to discover the spot price for a token pair. To discover the spot price for a token pair, the DvAMM protocol layer is designed to have an oracle that feeds it the spot price for a token pair. An initial attempt was made to integrate chainlink oracles to provide the spot price for ETH/USDC, the token pair currently implemented on the exchange. However, a real oracle providing real-time spot price data feeds was not integrated because each request to an oracle required payment in

cryptocurrency. Therefore, a mock oracle was designed and implemented to feed mock spot prices.

3.3.5 Challenges

Ethereum smart contract development is mainly done using the programming languages Vyper, Solidity and Yul (41). The smart contracts for the DvAMM protocol layer are designed with Solidity as the intended programming language. The smart contracts are designed using Solidity since it has a larger development community (42) and a lot of learning resources readily available. Designing and implementing smart contracts in Solidity raises a few design challenges. The challenges introduced are:

1. Arithmetic Precision: Solidity does not allow floating-point representation, and this causes precision and rounding errors (43). A perpetual exchange requires high precision and low rounding errors.
2. Solidity does not allow for string comparisons.
3. External functions cannot return struct types.

The challenges mentioned above should be considered, and the designed smart contracts should tackle these challenges appropriately when being developed.

4 Implementation

This chapter discusses how the designed perpetual swap exchange utilizing a dynamic vAMM was implemented. The researcher details the implementation of core components of the application and blockchain layer and the challenges faced during their development. This chapter further talks about the reasoning and tradeoffs behind implementation decisions. The proof-of-concept perpetual swap exchange is developed using a bottom-up approach where smaller functionalities or programs are developed individually, and these small functionalities or programs are integrated to form a wider system (44).

The code implementation for the designed perpetual swap exchange was built on top of a tutorial to build an AMM based exchange (45) (46). Section 4.1 details how the application layer and the application layer's components were implemented. Section 4.2 discusses the implementation of the smart contract-based blockchain or DvAMM protocol layer.

4.1 Application Layer

The application layer is developed using Javascript, React and Ethers. React is a javascript library used for building user interfaces (47). Ethers is a javascript library for interacting with the Ethereum Blockchain, and its ecosystem (48).
The implementation of various components forming the application layer is described in sections 4.1.1 and 4.1.2, respectively. Section 4.1.1 describes the implementation of the UI, and section 4.1.2 describes the implementation of BLIS.

4.1.1 User Interface (UI)

A user-friendly UI is implemented for the perpetual swap exchange. The UI is developed using a component-based architecture approach using react.js. Each component developed represents a different screen or tab of the UI. For example, the open trade tab in figure 4.1 is developed as a component in react.js. As mentioned in the design section, the UI consists of 2 separate views: the user view for the traders and the test view for the researcher. Figures 4.1 and 4.2 show the user view and test view implemented by the system.

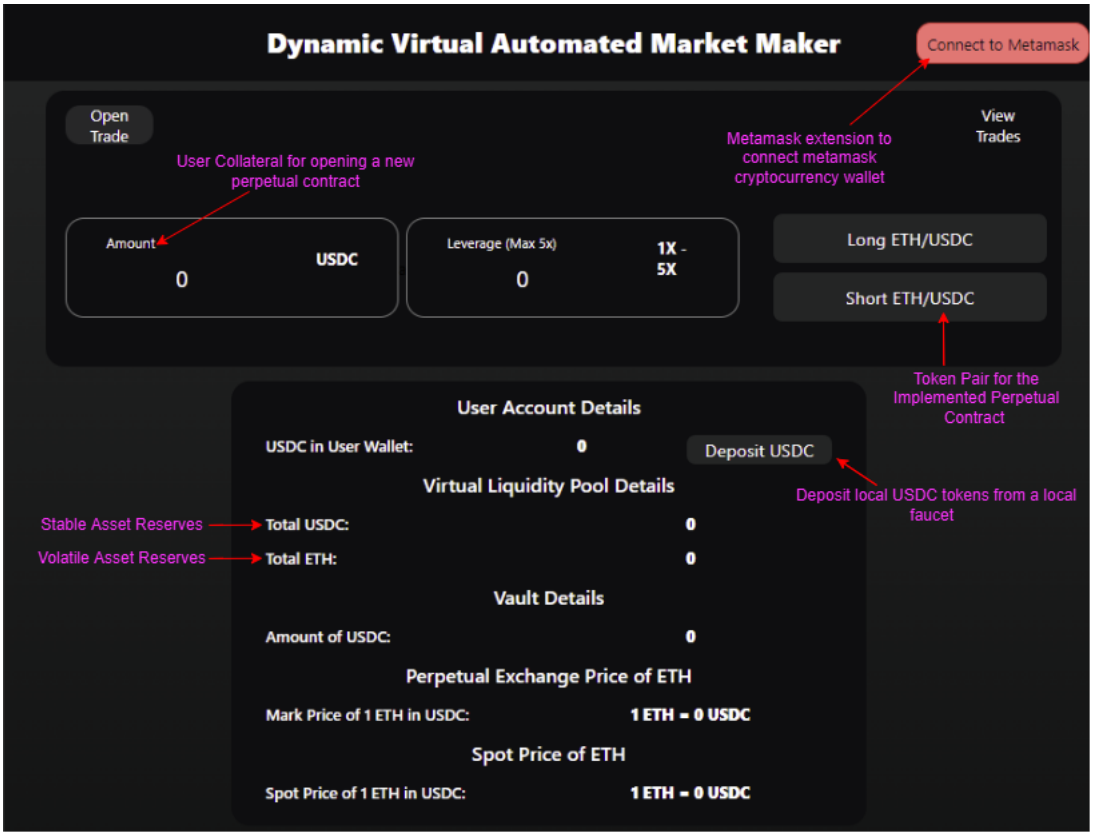

Figure 4.1: Perpetual Swap Exchange User View UI: Open Trade Tab

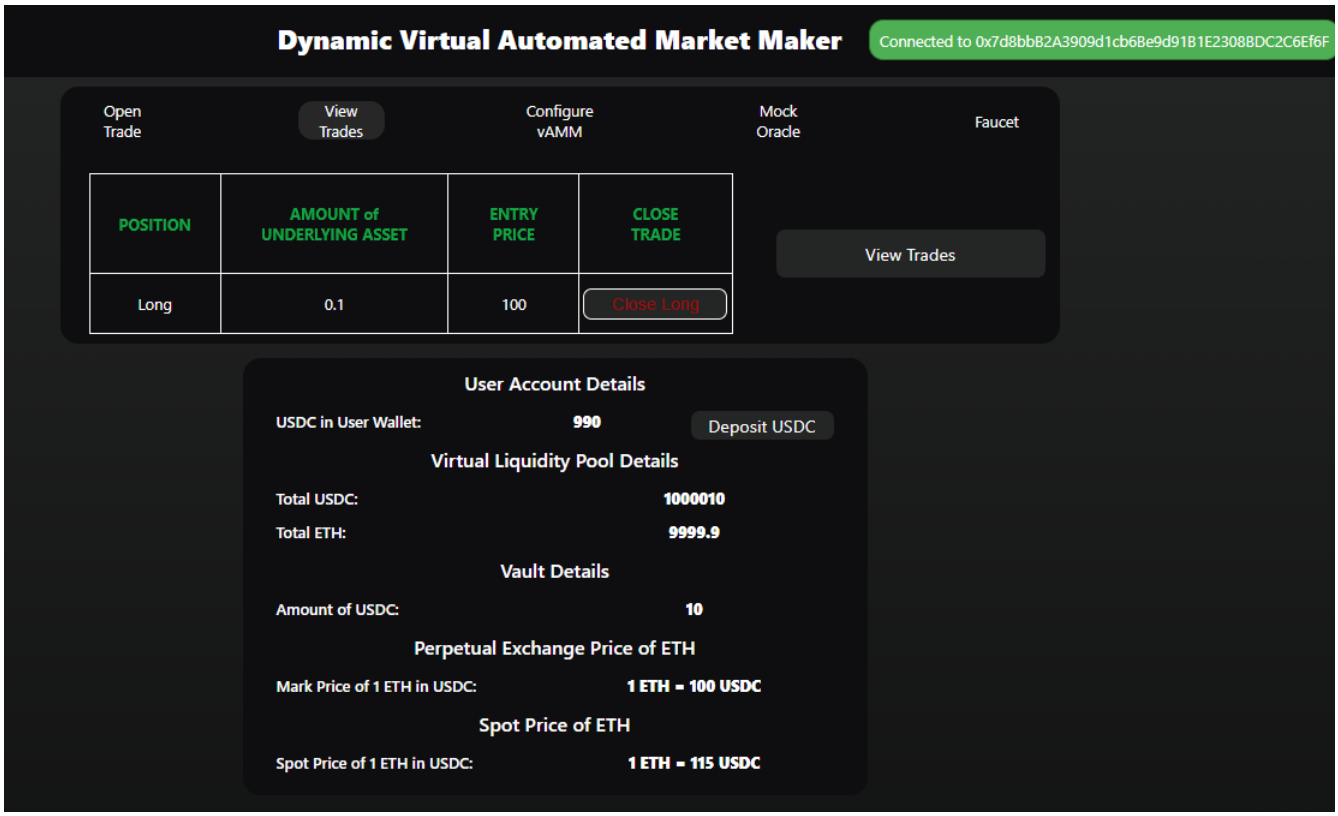

Figure 4.2: Perpetual Swap Exchange Test View UI: View Trade Tab

The user view seen in figure 4.1 provides an open trade tab and a view trade tab. The open trade tab enables a trader to open a trade by providing the collateral a trader wants to deposit and the leverage a trader wants on the perpetual contract. The view trade tab seen in figure 4.2 shows a trader their open perpetual contracts. The connect to metamask button allows traders to connect the perpetual exchange to their metamask browser blockchain wallet.

The current implementation of the perpetual swap exchange implements a local USDC token and does not use the ERC-20 USDC token. A local USDC token is implemented because it allows the researcher to have an unlimited token supply to perform experiments on the implemented system. The researcher does not need to find and request funds from external faucets to open and close trades on the system.

A trader needs to deposit USDC into their local perpetual swap exchange account and use these funds as collateral to open or close perpetual contracts. A trader can deposit funds fusing the "Deposit USDC" button seen in figure 4.1. The "Deposit USDC" button opens up a modal displaying a faucet. The users can request local USDC tokens from this faucet.

The details component can be seen in figures 4.1 and 4.2. The details component displays essential information such as the current USDC balance in a trader's local exchange account, the state of the virtual liquidity pool, vault balance and spot and mark price for the ETH/USDC token pair.

The test view can be seen in figure 4.2. .The test view consists of additional features that make the experimentation process simpler for the researcher. The implemented test view provides 3 additional tabs compared to the user view. These three tabs are "Configure vAMM", "Mock Oracle" and "Faucet" and can be seen in figure 4.2. The mock oracle tab allows the researcher to feed a custom spot price for the token pair being exchanged on the perpetual swap exchange. The mock oracle tab can be seen in figure 4.3. Setting a custom spot price from the UI over manually setting a mock price in the smart contract code will provide some convenience in performing configuration changes on the vAMM.

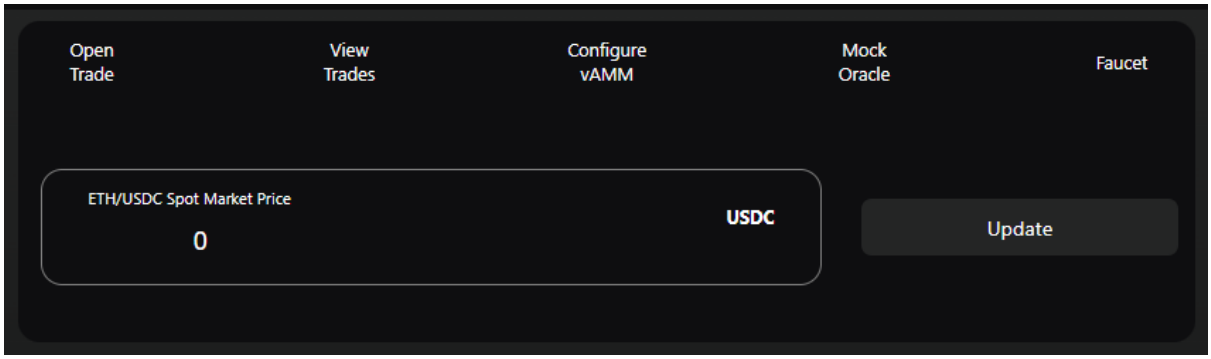

Figure 4.3: Perpetual Swap Exchange Test View UI: Mock Oracle Tab

The "Faucet" tab is implemented to allow the researcher to request the desired amount of USDC tokens directly along with the "Deposit USDC" button. The faucet tab can be seen in figure 4.4. The Faucet tab formulates a request to the smart contract code in the protocol layer for the requested amount of USDC.
The USDC tokens provided by the faucet do not have any real monetary value.

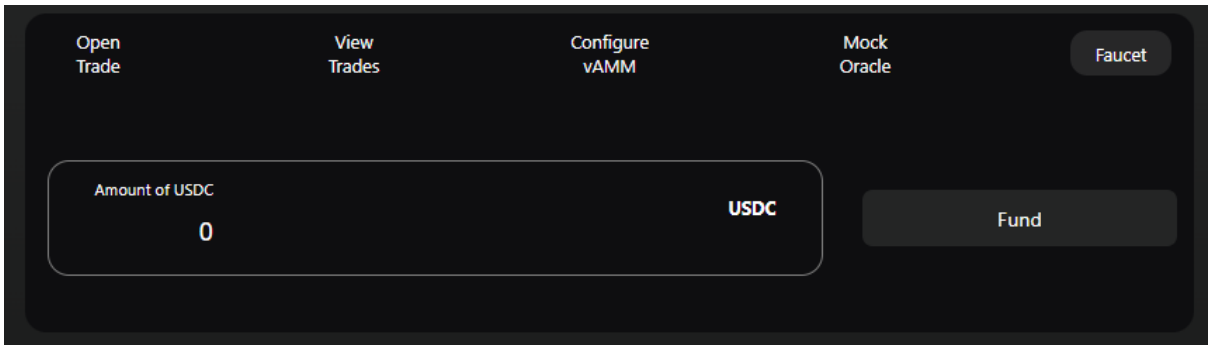

Figure 4.4: Perpetual Swap Exchange Test View UI: Faucet Tab

The implemented test view allows the researcher to change the configuration of the vAMM through the "Configure vAMM" tab. The purpose of configuring a vAMM is to adapt the mark price of a token pair to the spot price. The "Configure vAMM" tab seen in figure 4.5 allows the researcher to update the mark price of ETH/USDC. The researcher can update the mark price by either providing the new exchange rate of the token pair or by clicking on the button "Update Mark Price to Spot Price".

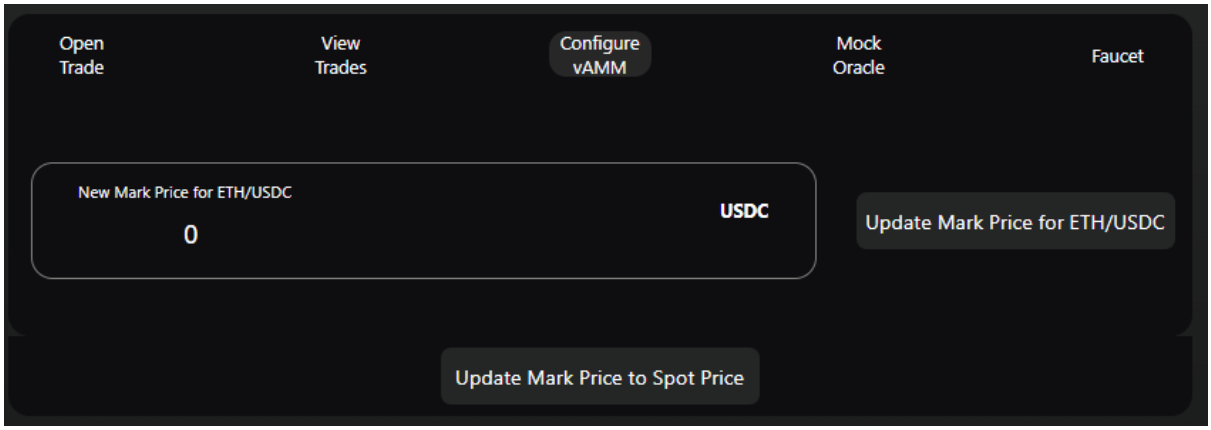

Figure 4.5: Perpetual Swap Exchange Test View UI: Configure vAMM Tab

The perpetual swap exchange being implemented only allows for opening and closing ETH/USDC perpetual contracts for this project. Perpetual markets for different token pairs can be created by initializing a new DvAMM for each desired token pair. However, due to time constraints, more perpetual markets were not implemented. Since this research aims to explore dynamic vAMMs and observe the effects of configuration changes on a vAMM, one perpetual contract or market will be sufficient to perform the study.

4.1.2 Blockchain Layer Interaction Service (BLIS)

BLIS is implemented for the application layer to communicate and interact with the smart contracts implemented in the DvAMM protocol layer. The application layer needs to communicate with the DvAMM protocol layer to carry out all the system's functionalities being developed.

BLIS is implemented using the design mentioned in section 3.2.2. BLIS is developed using ethers.js. Ethers.js enables BLIS to interact with smart contracts deployed on the blockchain. The smart contract code for the DvAMM protocol layer is deployed on the rinkeby ethereum testnet (49)

Ethers.js is a javascript library for interacting with the Ethereum Blockchain (48). The system implemented uses the contract interaction API provided by ethers.js to connect to the code deployed on the blockchain. The contract interaction API provided by ethers.js allows for a straightforward way to serialize calls to a smart contract deployed on a blockchain and to deserialize the results of the call to the smart contract (48). Ethers js provides a contract object which is an abstraction of the smart contract code that has been deployed to a blockchain. BLIS communicates with the smart contract based DvAMM protocol layer by creating an instance of the contract object provided by ethers.js and passes the following parameters to the instance of the contract object:

1. The object the address of the smart contract the application layer needs to connect to
2. The application binary interface or abi of the smart contract the application layer needs to connect to, and
3. The wallet address of the user attempting to connect with this smart contract

If an instance of the contract object is created successfully, ethers.js contract interaction API will return an abstraction of the smart-contract code that is deployed on the rinkeby testnet. The application layer can now interact with the smart contract methods that provide the functionalities of the DvAMM protocol layer.
BLIS connects the application layer to the DvAMM protocol layer smart contracts. The perpetual exchange only needs to interact with the clearinghouse smart contract to provide the functionality offered by the user view UI. The researcher needs to interact with the DvAMM smart contract directly to test various configurations on the vAMM. The code snippet that enables BLIS to connect to a smart contract deployed on the blockchain using ethers.js can be seen in figure 4.6

An exapmple of an ethers.js smart contract method call can be seen in figure 4.7

BLIS could also be implemented using Web3.js. Web3.js is a similar javascript library to ethers.js. Ethers.js was chosen as the preferred library for implementing BLIS due to better

```
signer = provider.getSigner();
add = await signer.getAddress();
setAddress(add);
try {
  const contract = new ethers.Contract(CONTRACT_ADDRESS, abi, signer);
```

Figure 4.6: BLIS connection process to a smart contract deployed on the blockchain

```
let response;
response = await props.contract.openShortPosition(amountUSDC * PRECISION, leverage);
await response.wait();
```

Figure 4.7: Ethers.js function call for opening a trade

web performance and faster loading times as compared to Web3.js (50). Ethers.js was also the implementation choice for BLIS due to the better quality of documentation and learning tutorials (50).

4.2 Blockchain or DvAMM Protocol Layer

This section provides an in-depth explanation of how the DvAMM protocol layer is implemented.

The smart contracts forming the DvAMM protocol layer were developed and deployed using Remix. Remix is an easy-to-use IDE that makes the process of developing, deploying, and administering smart contracts for Ethereum like blockchains simple (51). The smart contracts developed were first deployed to a local Ethereum blockchain provided by Ganache (52). The functionalities of the smart contracts were thoroughly tested on the local blockchain, following which the smart contracts were deployed to the rinkeby Ethereum testnet (49) using Remix IDE.

The smart contracts forming the DvAMM protocol layer are written using the programming language Solidity. The Solidity compiler version used to compile smart contracts is 0.8.13. The implemented smart contracts use the SafeMath library provided by OpenZeppelin. Solidity does not allow for floating-point literals, and all calculations are either performed using integers or fixed-point numbers (53). Therefore, Arithmetic operations in Solidity wrap on overflow and this can easily result in bugs in the smart contract code (54) The SafeMath library validates arithmetic operations to ensure that any arithmetic operation does not lead to an overflow or underflow.

The implementation of all components forming the DvAMM protocol layer is described between sections 4.2.1 and 4.2.5, respectively. Section 4.2.1 discusses the implementation of the Clearinghouse and its functionalities. Section 4.2.2 explains how the dynamic vAMM

was implemented and how the DvAMM protocol layer accepts and makes configuration changes to the dynamic vAMM. Section 4.2.3 describes how the smart contract implementing the Vault handles user collateral. Section 4.2.4 explains how the mock oracle provides mock spot prices to the DvAMM protocol layer.

4.2.1 Clearinghouse

The clearinghouse is responsible for the opening, closing, and recording perpetual contracts. The clearinghouse is also responsible for returning information about open perpetual contracts of a user and for returning details about the vault, token reserves in the virtual liquidity pool, the mark price and spot price for the ETH/USDC token pair.

The clearinghouse is implemented as a Solidity smart contract. Each functionality offered by the clearinghouse is implemented as a function in Solidity code. For example, the clearinghouse smart contract implements the openLongPosition method seen in figure 4.10 to open a new long position perpetual contract.

All the events and functions implemented by the clearinghouse smart contract can be seen in table 4.1. This section explains the logic and execution behind each function implemented by the clearinghouse smart contract. The clearinghouse stores the collateral amount for each new perpetual contract in the vault smart contract. The clearinghouse smart contract imports the smart contract code for the vault.

Table 4.1: Clearinghouse Functions and Events

Functions	openLongPosition(uint256, uint256)
	openShortPosition(uint256, uint256)
	closePosition(Position, uint256, uint256, uint256, uint256)
	getTrade(uint256)
	getPoolDetails()
	getMyHoldings
Events	PositionOpened(address, Position, uint256, uint256, uint256, uint256)
	PositionClosed(address, Position, uint256, uint256, uint256, uint256)

The clearinghouse smart contract imports the dynamic vAMM smart contract code as the clearinghouse needs to use the functionality provided by the dynamic vAMM to open and close perpetual contracts. The clearinghouse stores details of all open perpetual contracts in a struct called 'Trade'. The purpose of the struct 'Trade' Is to store contract specifications of each open perpetual contract. The position of each trade (long or short) is represented through an enum called 'Position'. A 'Position' can either be long or short. The struct 'Trade' and enum 'Position' can be seen in figure 4.8.

The clearinghouse stores information regarding open perpetual contract for each trader using

```
enum Position {
    LONG,
    SHORT
}

struct Trade {
    Position position; // Long or short
    uint256 collateral;
    uint256 tradeAmountInEth;
    uint256 entryPrice; //eth Price In Usdc On Contract Initiation
    uint256 leverage;
}
```

Figure 4.8: Clearinghouse struct Trade and enum Position

a mapping called 'allTradesofTrader'. 'allTradesofTrader' maps the address of a trader or a user to an array of the struct Trade. 'allTradesofTrader' can be considered a hashmap of key-value pairs where the key is the address of the trader. The values are the open perpetual contracts of the trader. The struct trade represents the open perpetual contracts. The mapping 'allTradesofTrader' can be seen in figure 4.9 The address of a trader is the address of the metamask wallet account a trader connects to the system.

```
mapping(address => uint256) userTokenUSDCBalance;
mapping(address => Trade[]) allTradesofTrader;
```

Figure 4.9: Clearinghouse mapping allTradesofTrader to map the address of a trader to an array of the Trades

The clearinghouse initialises an account for each trader on the perpetual swap exchange. This account stores USDC that a trader would like to use for trading on the exchange. A trader's balance on the perpetual swap exchange is stored using a mapping 'userTokenUSDCBalance'. 'userTokenUSDCBalance' maps a trader's address to their perpetual exchange USDC balance. The mapping 'userTokenUSDCBalance' can be seen in figure 4.9.

The current implementation of the clearinghouse maintains a record of a trader's USDC balance on the perpetual swap exchange itself and does not use the ERC-20 USDC token. This implementation choice was made because it allows the exchange to provide local tokens to open and close trades. Implementing a local token removes the complexity of finding an external faucet to request funds from. External faucets do not provide a large number of tokens, making it difficult to open trades with a large margin or collateral.

The main purpose of the clearinghouse is to open and close perpetual contracts. The clearinghouse implements 2 functions to open new perpetual contracts and 2 functions to close existing perpetual contracts.

4.2.1.1 Opening a perpetual contract

The functions that implement the functionality of opening new perpetual contracts are 'openLongPosition' and 'openShortPositon'. 'openLongPosition' is responsible for implementing the logic for opening a new long position perpetual contract. Similarly, 'openShortPositon' is responsible for is responsible for implementing the logic for opening a new short position perpetual contract. The 'openLongPosition' and 'openShortPositon' functions can be seen in figures 4.10 and 4.11.

```
function openLongPosition(uint256 userCollateralAmount, uint256 leverage)
    public
    returns (uint256)
{
    uint256 currEthInUSDC = DvAMM.markEthPriceInUSDC();
    uint256 userUSDCBalance = userTokenUSDCBalance[msg.sender];
    require(userUSDCBalance >= userCollateralAmount, "Low USDC balance");

    // Updating USDC balance of the user.
    userTokenUSDCBalance[msg.sender] = userTokenUSDCBalance[msg.sender].sub(
        userCollateralAmount
    );

    // Updating the vault
    updatedVaultBalance = Vault.deposit(userCollateralAmount, msg.sender);

    // Calculating total contract value with leverage
    uint256 vUSDCwithLeverage = userCollateralAmount * leverage;
    uint256 vETHpurchased = DvAMM.updateVAMMOpenLongPosition(
        vUSDCwithLeverage
    );

    Trade memory trade = Trade( Position.LONG, userCollateralAmount, vETHpurchased,
                                currEthInUSDC, leverage);
    allTradesofTrader[msg.sender].push(trade);

    vEthLong[msg.sender] = vEthLong[msg.sender] + vETHpurchased;
    longPositionLiquidity = longPositionLiquidity + vETHpurchased;

    emit PositionOpened(msg.sender, Position.LONG, vETHpurchased, updatedVaultBalance,
        currEthInUSDC, allTradesofTrader[msg.sender].length
    );

    return vETHpurchased;
}
```

Figure 4.10: Clearinghouse method to open a long perpetual contract

openLongPosition is implemented to handle the case when a trader submits a request with collateral and leverage to open a new long position perpetual contract. The function openLongPosition in the clearinghouse first checks if the trader has enough USDC balance to deposit the collateral for opening a long perpetual contract. If the trader has enough balance, the collateral or margin amount for the trade is deducted from the user's local USDC account. The trade's collateral or margin amount is then deposited in the Vault. Once the collateral has been deposited, openLongPosition calculates the total contract value of the long perpetual contract being opened.

The total contract value of a perpetual contract is the collateral amount times leverage. The leverage for each trade is set to 1 by default. In the current implementation, the maximum leverage accepted for a trade is 5X.
The total contract value is in virtual USDC (vUSDC) tokens, and it is passed to the dynamic vAMM to return the number of virtual ETH (vETH) tokens a trader has bet on. When a trader opens a long position, the trader is fundamentally betting on the price of ETH to rise in the future. openLongPosition calls the function 'updateVAMMOpenLongPosition' to receive the total number of vETH tokens a trader has bet on based on the total long position contract value. The 'updateVAMMOpenLongPosition' is provided by the dynamic VAMM smart contract and is imported by the clearinghouse. The working of 'updateVAMMOpenLongPosition' is discussed in detail in section 4.2.2.

After receiving the total vETH tokens that a trader has opened a long position on, openLongPosition creates a 'Trade' object with the perpetual contract details. The newly created 'Trade' object is then pushed to the 'allTradesofTrader' mapping, which stores the open perpetual contracts for a trader.

openLongPosition stores the total liquidity a trader has locked in long position perpetual contracts. openLongPosition also updates the total liquidity locked in the perpetual swap exchange for long position perpetual contracts. An event PositionOpened is emitted at the end of the function. Event PositionOpened is emitted to store a record of the contract specifications on the blockchain. The function returns that amount vETH tokens that the contract is opening a position on to communicate that the perpetual contract has been opened.

The clearinghouse implements the function openShortPositon to handle requests for opening short perpetual contracts. The logic and sequence of actions that the function openShortPositon implements is identical to the function openLongPosition.

When a trader opens a short position, a trader is essentially betting that the price of ETH is bound to fall from its current valuation in the future. openShortPositon calls the function 'updateVAMMOpenShortPosition' to receive the total number of vETH tokens a trader has bet on based on the total short position contract value. The

```
// Calculating total contract value with leverage
uint256 vUSDCwithLeverage = userSellAmountInUSDC * leverage;
uint256 vETHsold = DvAMM.updateVAMMOpenShortPosition(vUSDCwithLeverage);

Trade memory trade = Trade(
    Position.SHORT,
    userCollateralAmount,
    vETHsold,
    currEthInUSDC,
    leverage
);
allTradesofTrader[msg.sender].push(trade);

vEthShort[msg.sender] = vEthShort[msg.sender] + vETHsold;
shortPositionLiquidity = shortPositionLiquidity + vETHsold;
```

Figure 4.11: Clearinghouse method to open a short perpetual contract

'updateVAMMOpenShortPosition' seen in figure 4.11 is provided by the dynamic VAMM smart contract and imported by the clearinghouse. The working of 'updateVAMMOpenLongPosition' is discussed in detail in section 4.2.2.

Like the function openLongPosition, openShortPositon creates and pushes a 'Trade' object to the mapping 'allTradesofTrader'. openShortPosition then updates the liquidity a trader has locked in short position perpetual contracts and the total liquidity locked by the perpetual swap exchange in short position perpetual contracts. An event PositionOpened is emitted to store a record of the contract specifications on the blockchain. Finally, the function returns the amount of vETH tokens that the short contract is opening a position on to communicate that the perpetual contract has been opened.

4.2.1.2 Closing a perpetual contract

The function that implements the functionality of closing a perpetual contract is called 'closePosition.' The function closePosition is responsible for implementing the logic for an existing long or short perpetual contract. The code for the closePosition method can be seen in figures 4.12 and 4.13.

The closePosition function expects parameters regarding the perpetual contract that it needs to close out. The closePosition function applies the parameters passed to it to find the contract the trader has requested to close in the mapping 'allTradesofTrader'. If a matching perpetual contract is found in the mapping 'allTradesofTrader', closePosition first checks the position of the contract the trader has requested to close. Based on the position of the

```
function closePosition( Position _position, uint256 _collateral, uint256 _closePositionAmountETH, uint256 _entryPriceInUSDC,
                        uint256 leverage)
    public returns (uint256)
    {
    require(allTradesofTrader[msg.sender].length > 0, "No trades");
    for (uint256 i = 0; i < allTradesofTrader[msg.sender].length; i++) {
        if (
            allTradesofTrader[msg.sender][i].position == _position &&
            allTradesofTrader[msg.sender][i].collateral == _collateral &&
            allTradesofTrader[msg.sender][i].tradeAmountInEth == _closePositionAmountETH &&
            allTradesofTrader[msg.sender][i].entryPriceInUSDC == _entryPriceInUSDC &&
            allTradesofTrader[msg.sender][i].leverage == leverage
        ) {
            if (_position == Position.LONG) {
                uint256 contractValuevUSDC = DvAMM.updateVAMMCloseLongPosition(_closePositionAmountETH);
                uint256 actualUserUSDCWithoutLeverage = contractValuevUSDC / leverage;

                withdrawUserReturn = Vault.withdraw(actualUserUSDCWithoutLeverage, msg.sender);

                userTokenUSDCBalance[msg.sender] = userTokenUSDCBalance[msg.sender].add(withdrawUserReturn);

                deleteTradeofTrader(i);
                vEthLong[msg.sender] = vEthLong[msg.sender] - _closePositionAmountETH;

                longPositionLiquidity = longPositionLiquidity -_closePositionAmountETH;

                emit PositionClosed(msg.sender, Position.LONG, _closePositionAmountETH, actualUserUSDCWithoutLeverage,
                    _entryPriceInUSDC, allTradesofTrader[msg.sender].length
                );

                return withdrawUserReturn;
            }
```

Figure 4.12: Clearinghouse method to open a close a perpetual contract.

contract that needs to be closed, relevant code blocks are executed to close that contract. Figure 4.12 shows the smart contract code for closing a long position perpetual contract and figure 4.13 shows the code for closing a short position perpetual contract.

If the trader has requested the closing of a long position perpetual contract, the closePosition method calls the function 'updateVAMMCloseLongPosition' seen in figure 4.13 to receive the return value of the perpetual contract. The return value of the perpetual contract is received as vUSDC tokens. The vUSDC tokens indicate the return value of the contract in actual USDC tokens, and this value is stored in the variable 'contractValuevUSDC' seen in figures 4.12. The amount of USDC to be returned to the user is calculated by value dividing 'contractValuevUSDC' with the leverage value of the contract being close. After calculating the amount of USDC to be returned to the user, this amount is withdrawn from the vault. The withdrawn amount from the vault is then added to the trader's local USDC exchange account.

After depositing the contract return value to the trader's local account, the perpetual contract being closed is deleted from the mapping 'allTradesofTrader'. The total liquidity the trader closing the perpetual contract has locked in long positions is updated. The perpetual swap exchange's total liquidity locked in long positions is also updated. An event PositionClosed is emitted to store a record of the closed contract on the blockchain. Finally, the function returns the amount of USDC withdrawn from the vault to communicate that

```
if (_position == Position.SHORT) {
    uint256 vUSDCreturnedOnClosingShort = DvAMM.updateVAMMCloseShortPosition(_closePositionAmountETH);

    uint256 actualUserUSDCWithoutLeverage = USDCreturnedOnClosingShort / leverage;

    uint256 contractReturnsAndCollateral = 0;
    uint256 _userCollateralAmount = allTradesofTrader[msg.sender][i].collateral;

    if (actualUserUSDCWithoutLeverage > _userCollateralAmount) {
        // The user has experienced a loss on short contract closure.
        contractReturnsAndCollateral = _userCollateralAmount - (actualUserUSDCWithoutLeverage -_userCollateralAmount);
    } else {
        // The user has experienced a profit on short contract closure.
        contractReturnsAndCollateral = _userCollateralAmount + (_userCollateralAmount - actualUserUSDCWithoutLeverage);
    }

    withdrawUserReturn = Vault.withdraw( contractReturnsAndCollateral, msg.sender);

    userTokenUSDCBalance[msg.sender] = userTokenUSDCBalance[msg.sender].add(contractReturnsAndCollateral);

    deleteTradeofTrader(i);
    vEthShort[msg.sender] = vEthShort[msg.sender] - _closePositionAmountETH;

    shortPositionLiquidity = shortPositionLiquidity - _closePositionAmountETH;

    emit PositionClosed(msg.sender, Position.SHORT, _closePositionAmountETH, withdrawUserReturn, _entryPriceInUSDC,
        allTradesofTrader[msg.sender].length );

    return contractReturnsAndCollateral;
}
```

Figure 4.13: Clearinghouse code block to implement closing a short perpetual contract

the long position perpetual contract is closed.

The logic implemented to close a short position perpetual contract is identical to that implemented to close a long position perpetual contract. Figure 4.13 shows the smart contract code executed to close a short position.

4.2.1.3 Viewing a Trader's Open Perpetual Contracts

The clearinghouse is responsible for returning information about open perpetual contracts of a trader. The clearinghouse method that implements the functionality of returning details of open perpetual contracts of a trader is called 'getTrade'. The smart contract code for the method 'getTrade' can be seen in figure 4.14.

The information of all open trades for a trader is stored in the mapping 'allTradesofTrader'. The information for each trade of a trader can be accessed individually by passing in the trade index.
The getTrade method returns the details of one open perpetual contract for a trader. getTrade takes in the index of a trade as a parameter. The getTrade method then uses this index to extract information from the mapping 'allTradesofTrader' and returns it to the caller.

The application layer calls the getTrade method in a loop to fetch information about all the open perpetual contracts of a trader. The application layer loops from index 0 to the total

```
function getTrade(uint256 tradeIndex)
    external
    view
    returns (uint8, uint256, uint256, uint256, uint256)
{
    require(
        allTradesofTrader[msg.sender].length > 0,
        "User has no open trades"
    );
    require(
        tradeIndex < allTradesofTrader[msg.sender].length,
        "Index too big"
    );
    Position tradePosition = allTradesofTrader[msg.sender][tradeIndex]
        .position;
    uint256 tradeCollateral = allTradesofTrader[msg.sender][tradeIndex]
        .tradeCollateral;
    uint256 tradeAmountEth = allTradesofTrader[msg.sender][tradeIndex]
        .tradeAmountInEth;
    uint256 tradeEntryPrice = allTradesofTrader[msg.sender][tradeIndex]
        .entryPrice;
    uint256 tradeLeverage = allTradesofTrader[msg.sender][tradeIndex].leverage;

    return (uint8(tradePosition), tradeCollateral, tradeAmountEth, tradeEntryPrice, tradeLeverage);
}
```

Figure 4.14: Clearinghouse 'getTrade' method to return details of an open perpetual contract.

number of trades -1 to fetch information regarding all the open perpetual contracts for a user.

The function getTrade is implemented in such a manner because solidity does not allow returning an array of structs. The method is implemented to only return details of one trade at a time to work around this limitation. The application layer can then call the getTrade method in a loop for getting details of each open perpetual contract.

4.2.1.4 Viewing System Information

The clearinghouse is responsible for returning details about the vault, token reserves in the virtual liquidity pool, the mark price and spot price for the ETH/USDC token pair.

The clearinghouse function implemented to return these details is called 'getPoolDetails'. The function 'getPoolDetails' returns the number of virtual USDC and virtual ETH tokens, the balance of the vault, the mark price of the ETH/USDC token pair and the spot price of ETH/USDC. The smart contract code for the method 'getPoolDetails' can be seen in figure 4.15.

```
function getPoolDetails()
    external
    view
    returns (
        uint256,
        uint256,
        uint256,
        uint256,
        uint256
    )
{
    uint256 vUSDCreserve = DvAMM.getvUSDCreserve();
    uint256 vETHreserve = DvAMM.getvETHreserve();
    uint256 vaultBalance = Vault.getBalance();
    uint256 markprice_ethInUSDC = DvAMM.markEthPriceInUSDC();
    uint256 spotPrice_ethInUSDC = mockOracle.getSpotPriceETHinUSDC();

    return (
        vUSDCreserve,
        vETHreserve,
        vaultBalance,
        markprice_ethInUSDC,
        spotprice_ethInUSDC,

    );
}
```

Figure 4.15: Clearinghouse 'getPoolDetails' method.

4.2.1.5 Viewing the USDC balance in a trader's local perpetual exchange account

The clearinghouse is responsible for returning the USDC balance in a trader's local perpetual exchange account. The clearinghouse function to return a trader's USDC balance is called 'getMyHoldings'. The smart contract code for the method 'getMyHoldings'can be seen in figure 4.16.

4.2.1.6 Fetching the total liquidity locked in long and short positions in implemented system

The clearinghouse implements logic to return the total liquidity locked in long and short positions in the implemented perpetual swap exchange. The function responsible for returning the total liquidity locked by the exchange is called

```
// Returns the USDC balance deposited by the user in the
// perpetual swap exchange account
function getMyHoldings() external view returns (uint256) {
    uint256 amountTokenUSDC = userTokenUSDCBalance[msg.sender];
    return amountTokenUSDC;
}
```

Figure 4.16: Clearinghouse method to get the USDC balance in a trader's local perpetual exchange account.

'getTotalLiquidityInLongShortPosition'. The smart contract code for the method 'getTotalLiquidityInLongShortPosition'be seen in figure 4.17.

```
function getTotalLiquidityInLongShortPosition()
    external
    view
    returns (uint256, uint256)
{
    return (longPositionLiquidity, shortPositionLiquidity);
}
```

Figure 4.17: Clearinghouse method total liquidity locked in long and short positions.

4.2.2 Dynamic vAMM

The Dynamic vAMM implemented is configurable. The implemented dynamic vAMM smart contract is responsible for:

1. Providing the number of vETH tokens a contract has opened a long position on. In simpler terms, providing the total number of vETH tokens that a trader has bet on based on the total contract value.
2. Providing the number of vUSDC tokens when a perpetual contract is closed. The amount of vUSDC tokens represents the value of the perpetual contract when it was closed.
3. Updating the virtual liquidity pool whenever a perpetual contract is opened or closed on the implemented perpetual swap exchange.
4. Providing the number of vETH and vUSDC tokens in the virtual liquidity pool.
5. Providing the mark price of the ETH/USDC token pair.

6. Configuring the dynamic vAMM to adjust the mark price for the ETH/USDC token pair on the perpetual swap exchange. The DvAMM adjusts to an indicated mark price by changing the value of the constant k in equation 1 .

The dynamic vAMM is implemented as a Solidity smart contract. Each functionality offered by the dynamic vAMM is implemented as a function in Solidity code. For example, the dynamic vAMM smart contract implements the updatePoolBalance method seen in figure 4.21 to update the virtual token reserves of the virtual liquidity pool.

All the functions implemented by the dynamic vAMM smart contract can be seen in table 4.2. Each section below explains the logic and execution behind each function implemented by the dynamic vAMM smart contract.

Table 4.2: Dynamic vAMM Functions

Functions	updatePoolBalance(uint256, uint256)
	updateVAMMOpenLongPosition(uint256)
	updateVAMMOpenShortPosition(uint256)
	updateVAMMCloseLongPosition(uint256)
	updateVAMMCloseShortPosition(uint256)
	getvUSDCreserve()
	getvETHreserve()
	markEthPriceInUSDC()
	updateMarkPriceToSpotPrice()
	updateMarkPriceUserInput(uint256)
	configureDvAMM(uint256, uint256)
Events	updatedMarkPrice(uint256, uint256)
	updatedPoolReservesConfigDvAMM(uint256, uint256)

4.2.2.1 Contract Initialization

The variables used in the dynamic vAMM smart contract and their initialization can be seen in figures 4.18 and 4.19

The dynamic vAMM smart contract imports the SafeMath library to perform arithmetic calculations. By design, the dynamic vAMM stores the virtual liquidity pool in it. The implemented virtual liquidity pool comprises virtual USDC (vUSDC) and virtual ETH (vETH) tokens.

The implemented dynamic vAMM smart contract stores the vUSDC and vETH token reserves in 2 variables called 'vUSDCreserve' and 'vETHreserve', as seen in figure vETHreserve. 'K' is an arithmetic constant equal to the product of the token reserves stored in the virtual liquidity pool. par The dynamic vAMM (DvAMM) implements a constant

```
contract DvAMM {
    using SafeMath for uint256;
    uint256 constant PRECISION = 1_000_000;
    uint256 vUSDCreserve;
    uint256 vETHreserve;
    uint256 K;
```

Figure 4.18: Dynamic vAMM smart contract variables.

product market maker (CPMM). Equation 1 shows the formula of the CPMM implemented by the dynamic vAMM smart contract.

$$vETHreserve * vUSDCreserve = K \tag{1}$$

```
constructor() {
    vUSDCreserve = 1000000 * PRECISION;
    vETHreserve = 10000 * PRECISION;
    K = vUSDCreserve.mul(vETHreserve);
}
```

Figure 4.19: Dynamic vAMM smart contract constructor to initialise variables.

For this study, the initial mark price for the ETH/USDC token pair is 1 ETH = 100 USDC. This mark price is chosen as it makes it simpler to understand calculations and updates to the virtual token reserves when perpetual contracts are opened or closed. Figure 4.19 shows the initialization values of the virtual token reserves. The virtual USDC token reserve is initialised to 1,000,000 tokens, and the virtual ETH token reserve is initialised to 10,000 tokens. The initial mark price can be derived from applying the formula in equation 2 and is equal to 1,000,000 divided by 10,000, which is equal to 100 USDC per ETH.

$$price = \frac{vUSDCreserve}{vETHreserve} \tag{2}$$

The initial token volumes are set to high values to lower the effect of opening perpetual

contracts with high collateral. For example, consider a virtual liquidity pool with 100 virtual USDC tokens (vUSDCreserve =100) and 1 virtual ETH token (vETHreserve = 1). The current mark price of this market is 1 ETH = 100 USDC. Let us assume that the spot price for ETH/USDC is the same as the mark price. If the dynamic vAMM receives a request to handle opening a trade with 90 vUSDC as collateral, the token reserves of the virtual liquidity pool will be greatly affected. Executing this trade would drastically change the mark price, and the delta between the spot and mark price for the token pair will be extremely high.

The token reserves are set to a high initial value to prevent such situations. As the volume of traders using the system grows, these values can be scaled up equally such that the mark price remains consistent.

Solidity does not support floating-point literals. To enable the system being implemented to handle floating-point numbers, a variable called 'PRECISION' is initialized. The variable 'PRECISION' can be seen in figures 4.18 and 4.19. 'PRECISION' is used to represent the fractional part of a number.

Every integer or number in the smart contract is multiplied by the precision value. The value of precision seen in fig 4.18 is six zeroes. This implies that each integer or number variable in the smart contract has an accuracy of up to six digits after the decimal point. For example, 100.123456 would be represented as 100123456 in Solidity.

The application layer stores the value of 'PRECISION' as well. Whenever the application layer receives an integer from the DvAMM protocol layer, it divides the value by precision to place a decimal place on the number. Figure 4.20 shows how the application layer code handles integers returned by the DvAMM protocol layer.

```
let response2 = await props.contract.getPoolDetails();
setTotalUSDC(response2[0] / PRECISION);
setTotalETH(response2[1] / PRECISION);
setTotalVault(response2[2] / PRECISION);
setEthPriceInUSD(response2[3].toNumber());
setSpotEthPriceInUSD(response2[4].toNumber());
```

Figure 4.20: Handling of floating point numbers in the Application layer.

4.2.2.2 Updating virtual liquidity pool token reserves

The dynamic vAMM is responsible for updating the token reserves of the virtual liquidity pool. The dynamic vAMM updates the virtual token reserves by implementing the method 'updatePoolBalance'. The smart contract code for 'updatePoolBalance' can be seen in

figure 4.21. updatePoolBalance uses the parameters passed to it to update the existing

```
function updatePoolBalance(uint256 newVUSDCreserve, uint256 newVETHreserve)
    internal
{
    vUSDCreserve = newVUSDCreserve;
    vETHreserve = newVETHreserve;
}
```

Figure 4.21: Dynamic vAMM 'updatePoolBalance' function .

vUSDC and vETH reserves.

4.2.2.3 Updating the dynamic vAMM - Opening new perpetual contract

The dynamic vAMM is responsible for updating the virtual liquidity pool and returning the value of the underlying asset, i.e. the number of vETH tokens that a trader has opened a contract on based on the contract's total value in USDC. The functions 'updateVAMMOpenLongPosition' and 'updateVAMMOpenShortPosition' implement the logic to update the virtual liquidity pool and return the number of vETH tokens when a trader opens either a long or short perpetual contract.

The smart contract code for updateVAMMOpenLongPosition and updateVAMMOpenShortPosition can be seeing in figures 4.22 and 4.22.

```
function updateVAMMOpenLongPosition(uint256 userBuyAmountUSDC)
    external
    returns (uint256)
{
    uint256 vUSDCreserveNew = vUSDCreserve + userBuyAmountUSDC;
    uint256 vETHreserveNew = K / vUSDCreserveNew;
    uint256 vETHboughtFromPool = vETHreserve - vETHreserveNew;

    updatePoolBalance(vUSDCreserveNew, vETHreserveNew);
    return vETHboughtFromPool;
}
```

Figure 4.22: Dynamic vAMM 'updateVAMMOpenLongPosition' method.

The method updateVAMMOpenLongPosition is called by the Clearinghouse smart contract when a trader requests to open a long perpetual smart contract. The function accepts an unsigned integer as its parameter. The unsigned integer represents the total contract value of the perpetual contract opened (in vUSDC tokens).

When a long perpetual contract is opened, a trader is fundamentally betting that the value of ETH is going to rise relative to USDC. A trader is willing to deposit collateral with leverage in USDC to bet on a certain number of ETH tokens.
Based on the current mark price of ETH/USDC and current token reserves in the system, the updateVAMMOpenLongPosition function helps return the number of vETH tokens that a trader has opened a long position on. If the value of ETH in the system rises after a perpetual contract is opened, a trader will earn a profit by closing this perpetual contract. Similarly, if the mark price of ETH in USDC falls after a perpetual contract is opened, a trader will incur a loss by closing this perpetual contract.

updateVAMMOpenLongPosition uses the total contract value (vUSDC) to update the vUSDC reserves of the liquidity pool. The new vETH reserves are then calculated by dividing the constant K by the new vUSDC reserves. The difference between the old vETH and new vETH reserves indicates the number of vETH tokens that a long perpetual contract is betting on. This value is stored in a variable called 'vETHboughtFromPool'.

The method updateVAMMOpenLongPosition also updates the virtual liquidity pool with the new vUSDC and vETH reserves by calling the method updatePoolBalance seen in figure 4.21. Finally, the value stored in the variable 'vETHboughtFromPool' is returned to the Clearinghouse.

The method updateVAMMOpenShortPosition is called by the Clearinghouse smart contract when a trader requests to open a short perpetual smart contract. The function accepts an unsigned integer as its parameter. The unsigned integer represents the total contract value of the perpetual contract opened in vUSDC tokens. The code for 'updateVAMMOpenShortPosition' can be seen in figure 4.23.

```
function updateVAMMOpenShortPosition(uint256 userSellAmountUSDC)
    external
    returns (uint256)
{
    uint256 vUSDCreserveNew = vUSDCreserve - userSellAmountUSDC;
    uint256 vETHreserveNew = K / vUSDCreserveNew;
    uint256 vETHsoldToPool = vETHreserveNew - vETHreserve;

    updatePoolBalance(vUSDCreserveNew, vETHreserveNew);
    return vETHsoldToPool;
}
```

Figure 4.23: Dynamic vAMM 'updateVAMMOpenShortPosition' method.

When a short perpetual contract is opened, a trader is fundamentally betting that the value of ETH is going to fall relative to USDC.

Let us consider a simple example to understand how shorting works. Trader 'A' has 3 ETH, and the current price for 1 ETH = 100 USDC. Trader 'A' sells the 3 ETH they own to a bank in exchange for 300 US dollars (USD), believing that the price of ETH is going to fall. 3 months later, trader 'A' wants to buy the 3 ETH they had sold to the bank. The price of 1 ETH had fallen to 90 USDC in the 3 months (mark price = 90 USD/ETH). Therefore, to buy 3 ETH again, the trader needs to exchange 270 USD (3 ETH X 90 USDC) with the bank selling ETH.
Trader 'A' fundamentally made a profit of 30 USDC on the 3 ETH because if trader 'A' had held 3 ETH instead of selling it, the trader would have made a loss on the valuation of ETH 3 months into the future. Similarly, if the price of ETH had risen to 110 USD/ETH, trader 'A' would have made a loss since they sold ETH for a cheaper rate and bought it back for more.

The function updateVAMMOpenShortPosition works similarly to trader A's example above. Under the hood, updateVAMMOpenShortPosition gives vUSDC tokens to a trader from the virtual liquidity pool for selling it vETH. Therefore, the vUSDC reserves of the virtual liquidity pool are reduced by a value equal to that of the total contract value (product of collateral in USDC and leverage). The new vETH reserves are then calculated by dividing the constant K by the new vUSDC reserves. Since the pool gave away vUSDC tokens, the number of vETH tokens in the liquidity pool has increased. This is so because the product of vUSDC and vETH tokens has to always be equal to K.

The method updateVAMMOpenShortPosition then calculates the vETH tokens that a user has opened a short position on by subtracting the new vETH reserves by the old vETH reserves. This value is also equivalent to the number of vETH tokens added to the virtual liquidity pool and is stored in the variable 'vETHsoldToPool' inside the method.

The method updateVAMMOpenShortPosition finally updates the virtual liquidity pool reserves and returns the value stored in 'vETHsoldToPool' to the Clearinghouse.

4.2.2.4 Updating the dynamic vAMM - Closing a perpetual contract

The dynamic vAMM is responsible for updating the token reserves of the virtual liquidity pool whenever a perpetual contract is closed. The dynamic vAMM is also responsible for returning the contract's total value in USDC when the perpetual contract is closed. The total value of a contract is derived from the number of vETH tokens that a contract had opened a position on and the current mark price of the ETH/USDC token pair. The functions' updateVAMMCloseLongPosition' and 'updateVAMMCloseShortPosition' implement the logic to update the virtual liquidity pool and return the number of vUSDC tokens representing the value of the contract when a trader closes a long or short perpetual contract. The smart contract code for updateVAMMCloseLongPosition and

updateVAMMCloseShortPosition can be seeing in figures 4.24 and 4.25.

```
function updateVAMMCloseLongPosition(uint256 closePositionAmountETH)
    external
    returns (uint256)
{
    uint256 vETHreserveNew = vETHreserve + closePositionAmountETH;
    uint256 vUSDCreserveNew = K / vETHreserveNew;
    uint256 vUSDCsoldPool = vUSDCreserve - vUSDCreserveNew;

    updatePoolBalance(vUSDCreserveNew, vETHreserveNew);
    return vUSDCsoldPool;
}
```

Figure 4.24: Dynamic vAMM 'updateVAMMCloseLongPosition' method.

The method updateVAMMCloseLongPosition is called by the Clearinghouse smart contract when a trader requests to close an open long position perpetual smart contract. The function accepts the number of vETH tokens that the contract had 'longed' on.

Closing a long perpetual contract is analogous to selling or adding the vETH tokens a contract had bet on back to the liquidity pool.

updateVAMMCloseLongPosition takes the vETH that tokens the long perpetual contract had bet on and them back to the vETH reserve of the virtual liquidity pool. The new vUSDC reserves are then calculated by dividing the constant K by the new vETH reserves. Since the pool gained vETH tokens, the number of vUSDC tokens in the liquidity pool has decreased. This is so because the product of vUSDC and vETH tokens must always be equal to K.

updateVAMMCloseLongPosition then finds the total value of the vETH tokens added back to the pool by finding the difference between the old and new vUSDC reserves. This value is also equivalent to the total contract value when closing the long perpetual contract.

The method updateVAMMCloseLongPosition finally updates the virtual liquidity pool reserves and returns the value of the closed contract to the Clearinghouse.

The method updateVAMMCloseShortPosition is called by the Clearinghouse smart contract when a trader requests to close an open short position perpetual smart contract. The function updateVAMMCloseShortPosition accepts the number of vETH tokens that the contract had 'shorted' on.
The smart contract code for the function updateVAMMCloseShortPosition can be seen in figure 4.25

Opening a short contract is analogous to selling a security (here vETH tokens) to the virtual liquidity pool. Closing a short perpetual contract is analogous to "buying back" or taking

```
function updateVAMMCloseShortPosition(uint256 closePositionAmountETH)
    external
    returns (uint256)
{
    uint256 vETHreserveNew = vETHreserve - closePositionAmountETH;
    uint256 vUSDCreserveNew = K / vETHreserveNew;
    uint256 vUSDCsoldPool = vUSDCreserveNew - vUSDCreserve;

    updatePoolBalance(vUSDCreserveNew, vETHreserveNew);
    return vUSDCsoldPool;
}
```

Figure 4.25: Dynamic vAMM 'updateVAMMCloseShortPosition' method.

vETH tokens from the virtual liquidity pool.

updateVAMMCloseShortPosition takes the number of vETH tokens that the perpetual contract being closed had 'shorted' on and subtracts this value from the vETH reserve of the virtual liquidity pool. The new vUSDC reserves are then calculated by dividing the constant K by the new vETH reserves. Since the pool lost vETH tokens, the number of vUSDC tokens in the liquidity pool has increased. This is so because the product of vUSDC and vETH tokens must always be equal to K.

updateVAMMCloseShortPosition then finds the total value of the vETH tokens added back to the pool by finding the difference between the new and old vUSDC reserves. This value is also equivalent to the total contract value when closing the short perpetual contract.

The method updateVAMMCloseShortPosition finally updates the virtual liquidity pool reserves and returns the value of the closed short contract to the Clearinghouse.

4.2.2.5 Providing token reserve information and the mark price of ETH/USDC token pair

The dynamic vAMM implemented is responsible for providing the number of vETH and vUSDC tokens in the virtual liquidity pool. The dynamic vAMM function to return the vUSDC token reserve in the virtual liquidity pool is called 'getvUSDCreserve'. The dynamic vAMM function to return the vETH token reserve in the virtual liquidity pool is called 'getvETHreserve'. The smart contract code for the functions 'getvUSDCreserve' and 'getvETHreserve' can be seen in figure 4.26

The dynamic vAMM implemented is also responsible for providing the mark price of the ETH/USDC token pair. The dynamic vAMM function to return the mark price of 1ETH in USDC is called 'markEthPriceInUSDC'. The smart contract code for the function 'markEthPriceInUSDC' can be seen in figure 4.27

```
function getvUSDCreserve() public view returns (uint256) {
    return vUSDCreserve;
}

function getvETHreserve() public view returns (uint256) {
    return vETHreserve;
}
```

Figure 4.26: Dynamic vAMM 'getvUSDCreserve' and 'getvETHreserve' functions.

```
function markEthPriceInUSDC() public view returns (uint256) {
    uint256 ethPriceInUSD = vUSDCreserve / vETHreserve;
    return ethPriceInUSD;
}
```

Figure 4.27: Dynamic vAMM 'markEthPriceInUSDC' method.

4.2.2.6 Configuring the dynamic vAMM

The purpose of implementing a dynamic vAMM is so that it can be configured. A dynamic vAMM helps a perpetual swap exchange adjust the mark price for a token pair to its spot price or close to its spot price. A perpetual swap exchange is more adaptable to market conditions if it can adjust its own mark price closer to the spot price for a token pair.

The system implemented uses a dynamic vAMM to adjust the mark price of the implemented token pair (ETH/USDC) to the spot price. The dynamic vAMM adjusts the mark price for a given token pair by configuring the virtual liquidity pool's reserves for the token pair.
The perpetual swap exchange implements the ETH/USDC token pair. The mark price for the token pair can be found using the formula seen in equation 3.

The mark price of a token pair is adjusted by scaling the value of the vUSDC token reserves and the constant k seen in equation 4.

$$markPrice = \frac{vUSDCreserve}{vETHreserve} \tag{3}$$

$$vETHreserve * vUSDCreserve = K \tag{4}$$

The current DvAMM implementation updates the mark price by keeping the number of vETH tokens fixed and adjusting the number of vUSDC tokens. For example, the spot price is 50 USDC per ETH and mark price is 100 USDC per ETH. The number of vUSDC tokens

is 1000, vETH tokens are 10 and k = 10,000. To update the mark price to be equal to the spot price, we can use the formula seen in equation 5. new-vUSDCreserve = 10(vETH) * 50 (spot price) = 500. The new vUSDC reserve is 500, and the vETH reserve is 10. The mark price can be derived from equation 3 and is equal to 500 vUSDC/10vETH = 50 USDC/ETH. The new value of the constant K = 500*10 = 5000.

$$new\ vUSDCreserve = newMarkPrice * current\ vETHreserve \tag{5}$$

The dynamic vAMM implements the function 'updateMarkPriceToSpotPrice' to adjust the mark price of ETH/USDC to the spot price of ETH/USDC. The smart contract code for the function 'updateMarkPriceToSpotPrice' can be seen in figure 4.28.

```
function updateMarkPriceToSpotPrice() public {
    uint256 oldMarkPrice = markEthPriceInUSDC();
    uint256 spotPriceETHinUSDC = mockOracle.getSpotPriceETHinUSDC();
    require(spotPriceETHinUSDC > 0, "Amount cannot be zero!");

    oracleNewUSDCreserve = vETHreserve * spotPriceETHinUSDC;
    configureDvAMM(oracleNewUSDCreserve, vETHreserve);

    uint256 newMarkPrice = markEthPriceInUSDC();
    require(newMarkPrice == spotPriceETHinUSDC, "Incorrect Price update!");
    emit updatedMarkPrice(oldMarkPrice, newMarkPrice);
}
```

Figure 4.28: Dynamic vAMM 'updateMarkPriceToSpotPrice' method.

updateMarkPriceToSpotPrice first fetches the spot current spot price of the token pair. The function then finds the number of new vUSDC token reserves to update the mark price. Following this, the dynamic vAMM is configured by calling the function 'configureDvAMM'. configureDvAMM updates the vUSDC token reserves and the value of the arithmetic constant K. After configuring the virtual liquidity pool token reserves and K, an event called 'updatedPoolReservesConfigDvAMM' is emitted to store a log of the configuration changes of the dynamic vAMM.

The smart contract code for the function 'configureDvAMM' and event 'updatedPoolReservesConfigDvAMM' 'can be seen in figures 4.29 and 4.30.

After the DvAMM has been configured, the function updateMarkPriceToSpotPrice checks if the new mark price is equal to the spot price. If the new mark price and spot price are equal, an event 'updatedMarkPrice' is emitted to store a log of the old and new mark prices. The code for the event 'updatedMarkPrice' can be seen in figure 4.30

The dynamic vAMM also implements a function to update the mark price of the

```
function configureDvAMM(uint256 newvUSDCResv, uint256 newvETHResv)
    internal
{
    require(newvUSDCResv > 0, "Amount cannot be zero!");
    require(newvETHResv > 0, "Amount cannot be zero!");

    vUSDCreserve = newvUSDCResv;
    vETHreserve = newvETHResv;
    K = vUSDCreserve.mul(vETHreserve);

    emit updatedPoolReservesConfigDvAMM(vUSDCreserve, vETHreserve, K);
}
```

Figure 4.29: Dynamic vAMM 'configureDvAMM' method.

```
event updatedMarkPrice(uint256 oldMarkPrice, uint256 newMarkPrice);

event updatedPoolReservesConfigDvAMM(
    uint256 newvUSDCResv,
    uint256 newvETHResv,
    uint256 newK
);
```

Figure 4.30: Dynamic vAMM 'updatedPoolReservesConfigDvAMM' and 'updatedMarkPrice' events.

ETH/USDC token pair to a user-provided price. The function to implement this is called 'updateMarkPriceUserInput' and the smart contract code for this function can be seen in figure 4.31.

The logic implemented by the function 'updateMarkPriceUserInput' is the same as the method 'updateMarkPriceToSpotPrice'. The only difference between the two functions is that one function updates the mark price to be equal to the provided user input, and one function updates the mark price to be equal to the spot price.

4.2.3 Vault

The Vault is responsible for securely holding the collateral deposits for all open perpetual contracts. This section explains the logic and execution behind each function implemented by the Vault.

The current implementation of the Vault maintains a record of the total number of USDC tokens locked in open perpetual contracts on the perpetual swap exchange itself. The

```
function updateMarkPriceUserInput(uint256 newMarkPrice) public {
    require(newMarkPrice > 0, "Amount cannot be zero!");
    uint256 oldMarkPrice = markEthPriceInUSDC();

    uint256 newvUSDCResv = vETHreserve * newMarkPrice;
    configureDvAMM(newvUSDCResv, vETHreserve);

    uint256 newMarkPriceAfterConfig = markEthPriceInUSDC();
    require(
        newMarkPriceAfterConfig == newMarkPrice,
        "Incorrect Price update!"
    );
    emit updatedMarkPrice(oldMarkPrice, newMarkPrice);
}
```

Figure 4.31: Dynamic vAMM 'updateMarkPriceUserInput' function.

perpetual swap exchange implements a local USDC token and does not use the ERC-20 USDC token. The reasoning behind this implementation choice can be found in sections 4.1.1 and 4.2.1.

The Vault is implemented as a Solidity smart contract. The Vault implements two functions and two events. All the events and functions implemented by the vault smart contract can be seen in table 4.3.

Table 4.3: Vault Functions and Events

Functions	deposit(uint256, address)
	withdraw(uint256, address)
Events	vaultDeposit(address, uint256)
	vaultWithdraw(address, uint256)

The vault smart contract is responsible for depositing and securely storing the collateral for all open perpetual contracts. The function that handles the new perpetual contract collateral deposits into the USDC vault is called 'deposit'. The implementation for the function 'deposit' can be seen in figure 4.32.

The vault smart contract is responsible for securely withdrawing and returning a trader's collateral along with the profit and loss incurred when a perpetual contract is closed. The function that handles the withdrawal of USDC tokens from the USDC vault is called

'withdraw'. The implementation for the function 'withdraw' can be seen in figure 4.32.

```
function deposit(uint256 userCollateralAmount, address trader) {
    require(userCollateralAmount > 0, "Collateral amount must be greater than 0");
    USDCvaultBalance += userCollateralAmount;
    emit vaultDeposit(trader, USDCvaultBalance);
}

function withdraw(uint256 actualUserUSDCWithoutLeverage, address trader) {
    require(actualUserUSDCWithoutLeverage > 0, "Withdraw amount must be greater than 0");
    require(actualUserUSDCWithoutLeverage <= USDCvaultBalance, "Insufficient balance");

    USDCvaultBalance -= actualUserUSDCWithoutLeverage;
    emit vaultWithdraw(trader, USDCvaultBalance);
}
```

Figure 4.32: Vault smart contract 'deposit' and 'withdraw' functions.

4.2.4 Mock Oracle

The mock oracle is responsible for providing functionality that allows the researcher to feed mock spot prices for the ETH/USDC token pair. The mock oracle is implemented as a Solidity smart contract. The mock oracle implements two functions and one event. All the events and functions implemented by the vault smart contract can be seen in table 4.4.

Table 4.4: Mock Oracle Functions and Events

Functions	getSpotPriceETHinUSDC()
	setSpotPrice(uint256)
Events	oraclePriceUpdatedEvent(uint256)

The mock oracle implemented is responsible for returning the spot price of the ETH/USDC token pair. The mock oracle is also responsible for setting the spot price of the ETH/USDC token pair based on the input price it receives. The mock oracle implements two functions called 'getSpotPriceETHinUSDC' and 'setSpotPrice' to fulfil these responsibilities. The implementation of these functions can be seen in figure 4.33.

The current mock oracle smart contract implementation stores the spot price of the ETH/USDC token pair in a variable called 'oracle1ETHinUSDC'. The method getSpotPriceETHinUSDC returns the value stored in the variable oracle1ETHinUSDC. The method setSpotPrice sets the value of the variable oracle1ETHinUSDC and returns the updated value stored in the variable oracle1ETHinUSDC.

```
function getSpotPriceETHinUSDC() public returns (uint256) {
    return oracle1ETHinUSDC;
}

function setSpotPrice(uint256 oracleSpotPrice) public returns (uint256) {
    require(
        oracleSpotPrice > 0,
        "Oracle spot price must be greater than 0"
    );
    oracle1ETHinUSDC = oracleSpotPrice;
    emit oraclePriceUpdatedEvent(oracle1ETHinUSDC);
    return oracle1ETHinUSDC;
}
```

Figure 4.33: Methods implemented by the mock oracle smart contract.

5 Evaluation

This chapter details the evaluation process for the implemented system. This chapter explains the methodology behind evaluating the dynamic vAMM based perpetual swap exchange. This chapter then provides the results of the evaluation performed and the discussion of these results.

This chapter evaluates the implemented system and discusses its strengths, limitations, and feasibility. This chapter suggests solutions to address the limitations of the implemented system and suggests useful metrics help and parameters that would automate the process of configuring a vAMM based on current market conditions.

Following the methodology, results and discussion sections, this chapter finally critically reviews the implemented DvAMM based perpetual swap exchange. Section 5.1 discusses the methodology behind evaluating the system. Section 5.2 provides the results obtained on evaluating the implemented system. Section 5.3 discusses the results obtained and section 5.4 critically reviews the implemented system.

5.1 Methodology

A series of experiments are performed to evaluate the implemented perpetual swap exchange and the dynamic vAMM. The experiments are performed to test the functionality and reliability of the implemented perpetual swap exchange. The experiments evaluate if the dynamic vAMM based perpetual exchange can adapt to external market conditions or not. This study implements a dynamic vAMM protocol to test if the protocol can adapt to current market conditions and to find the protocol's limitations.

To implemented system is evaluated by performing multiple sets of experiments with different initial parameter values. The parameter that is varied at the start of each set of experiments is the constant K seen in equation 1. Varying the value of K is equivalent to changing the values of the virtual liquidity pool reserves. The different sets of intial parameter values used to perform experiments on the system can be seen in tables 5.1 and 5.2

$$vETHreserve * vUSDCreserve = K \quad (1)$$

The values of K, vETH and vUSDC tokens are varied at the start of each set of experiments because these values impact the behaviour of the implemented system. The value of K with respect to the size of a trade determines the variation a trade causes on the virtual liquidity pool token reserves. If a trade vastly alters the token reserves of a virtual liquidity pool, the mark price of an exchange will diverge extremely from the spot price for that token pair. Therefore, the value of K and token reserves affect the trading experience for the users of the implemented system.

It is important to evaluate the system for different initial configurations to observe how the dynamic vAMM handles several scenarios. Performing experiments with different K values and token reserves will also allow this study to determine an optimal strategy to pick the K value.

The implemented system is evaluated based on how the dynamic vAMM behaves when the system is made to adapt to market conditions. Various experiments are performed on the system to observe how well the DvAMM based protocol can handle updates to the mark price for the implemented token pair (vETH/vUSDC). These experiments can be seen in 5.2. The experiments are performed to observe if trades can be liquidated once the mark price of a token pair is updated. The experiments also observe the impact on profits and losses of traders when the mark price of a token pair is updated.
External market conditions are provided as the spot price for ETH/USDC by the oracle to the system.

The experiments performed essentially update the mark price of ETH/USDC at different stages of trading on the perpetual swap exchange. Performing such experiments allows the researcher to observe the system's behaviour under various scenarios. For example, an experiment is performed to observe the system's behaviour when only 1 long perpetual contract is open, and the researcher updates the mark price of the system to be equal to the spot price.

The experiments evaluate the system under extreme scenarios (edge cases). Performing edge case experimentation allows this research to evaluate the performance and behaviour of a dynamic vAMM protocol under extreme scenarios. The dynamic vAMM protocol also needs to handle all possible cases that an AMM and vAMM protocol can handle.

The experiments seen in tables 5.1 are performed to test if the implemented dynamic vAMM based system can perform the role of a fully functional vAMM based perpetual exchange correctly. In the set of experiments seen in table 5.1, we do not update the mark price or token reserves manually. The experiments evaluate if the system performs correctly under

various scenarios. Each experiment in table 5.1 has an expected result or behaviour. If the implemented system does not yield results or behaviour equivalent to the expected results, it indicates flaws in the implemented system.

Tables 5.1 and 5.2 shows the list of experiments performed on the implemented DvAMM based perpetual exchange to evaluate it.

Table 5.1: Experiments to test if the implemented dynamic vAMM based system can perform as a fully functional vAMM based perpetual exchange.

No.	Experiment Description	Expected Results/Behaviour
Initial Parameters: vUSDC = 100,000, vETH = 1000, Mark Price 100 USDC, K = 100,000,000		
1.1	Open 3 long positions from one user account with random amounts of collateral for each trade. Close all long trades in any order	Long trades should open seamlessly. The vault balance should be updated. The dynamic vAMM calculations and exchange rates should be formulaically correct. All trades should close seamlessly. The system should not prevent the liquidation of any trades due to low funds in the vault. The vault balance should be 0 after all trades are closed.
1.2	Open 10 long trades from 5 trader accounts with random amounts of collateral for each trade. Close all long trades in any order	Long trades should open seamlessly The vault balance should be updated. The vAMM calculations and exchange rates should be formulaically correct. All trades should close seamlessly. The vault balance should be 0 after all trades are closed.
2.1	Open 3 short positions from one user account with random amounts of collateral for each trade. Close all short trades in any order.	Short trades should open seamlessly. The vault balance should be updated. The dynamic vAMM calculations and exchange rates should be formulaically correct. All trades should close seamlessly, and the system should not prevent the liquidation of any trades due to low funds in the vault. The Vault balance should be 0 after all trades are closed
2.2	Open 10 long trades from 5 trader accounts with random amounts of collateral for each trade. Close all long trades randomly.	

Table 5.1 continued from previous page

No.	Experiment Description	Expected Results/Behaviour
3	Open 10 long and 10 short trades in random order from 5 accounts with equal amounts of collateral for each trade (100 USDC). Close all trades in random order.	All contracts should open seamlessly. The vault balance should be updated. The dynamic vAMM calculations and exchange rates should be formulaically correct. All trades should close seamlessly. The system should not prevent the liquidation of any trades due to low funds in the vault
4.1	Total number of user accounts opening and closing trades = 5 Open Perpetual Contracts in the following order: Short trades = 5. (1 trade per account) Long trades = 10. (2 trades per account) Collateral for each trade = 100 USDC Liquidity in Longs >Liquidity in Short Closing Perpetual Contracts Close all short trades first followed by long trades after.	All contracts should open seamlessly. The vault balance should be updated with the opening of each contract. Dynamic vAMM calculations and exchange rates should be formulaically correct. All trades should close seamlessly. Since liquidity in long trades is greater than the liquidity locked in short trades and all short trades are closed first, all shorts should receive a loss on their trade collateral. All long trades should receive a profit on their trade collateral. The exchange should not prevent the liquidation of any trades due to low funds in the vault. The Vault balance should be 0 after all the trades are closed.
4.2	Simulate experiment 4.1 again and close all long trades first followed by short trades	Similar results as experiment 4.1should be observed. The only difference being that short traders should not experience losses on their trade collateral.

Table 5.1 continued from previous page

No.	Experiment Description	Expected Results/Behaviour
5.1	Total number of user accounts opening/closing trades = 5 Open Perpetual Contracts in the following order: Long trades = 5. (1 trade per account) Short trades = 10. (2 trades per account) collateral for each trade = 100 USDC Liquidity in Shorts >Liquidity in Longs Closing Perpetual Contracts: Close all long trades first followed by short trades.	All contracts should open seamlessly. The vault balance should be updated with the opening of each contract. The dynamic vAMM calculations and exchange rates should be formulaically correct. All trades should close seamlessly. Since the liquidity in short trades is greater than the liquidity in long trades and short trades are closed first, all long trades should receive a loss on their trade collateral. All shorts trades should receive a profit on their trade collateral. The exchange should not prevent the liquidation of any trades due to low funds in the vault. The Vault balance should be 0 after all the trades are closed.
5.2	Simulate experiment 5 again and close all short trades first followed by long trades	Similar results as experiment 5.1 should be observed. The only difference should be long traders should not experience losses on their trade collateral.

Table 5.2: Experiments to evaluate the behaviour of the DvAMM based system when the mark price is updated

No.	Experiment Description
Initial Parameters: Mark Price = 100 USDC/ETH Set 1. vUSDC = 1000 , vETH = 10, K = 10,000. Set 2. vUSDC = 100,000 , vETH = 1000, K = 100,000,000. Set 3. vUSDC = 10,000,000 , vETH = 100,000 , K = 1,000,000,000,000.	
1	Open 1 long position from one user account with collateral = 100 USDC. Note the mark price after the trade execution. Update Mark Price to be greater than the current mark price. Close the long position trade and observe the behaviour. Expected Result: The trader should receive a profit on trade closure since the mark price rose after the trader had opened a long position

Table 5.2 continued from previous page

No.	Experiment Description
2	Intialize system parameters (vETH, vUSDC) Open 1 short position from one user account with collateral = 100 USDC. Note the mark price after the trade execution. Update Mark Price to be less than the current mark price. Close the short position trade and observe the system's behaviour. Expected Result: The trader should receive a profit on trade closure since the value of ETH decreased in USDC after the trader had opened their trade.
3	Initialise system parameters (vETH, vUSDC) Open 1 long position from one user account with collateral = 100 USDC. Note the mark price after the trade execution. Update Mark Price to be less than the current mark price. Close the long position trade and observe the system's behaviour. Expected Result: The trader should receive a loss on trade closure since the mark price the value of ETH in USDC fell after the trader had opened a long position
4	Intialize system parameters (vETH, vUSDC) Open 1 short position from one user account with collateral = 100 USDC. Note the mark price after the trade execution. Update Mark Price to be greater than the current mark price. Close the short position trade and observe the system's behaviour. Expected Result: The trader should receive a loss on trade closure since the value of ETH increased in USDC after the trader had opened their trade.

Table 5.2 continued from previous page

No.	Experiment Description
5	Intialize system parameters (vETH, vUSDC) Open 10 long and 10 short positions from five user accounts. Each account will open 1 long trade and 1 short trade. Open long and short trades alternatively. The collateral of each trade is100 USDC. The mark price after all trades are open should be equal to the initial mark price. Update Mark Price to be greater than the current mark price. Close all positions in any order. Observe if the system can handle trade closures when the mark price is updated and the liquidity of long and short trades is the same. Repeat the experiment by reducing the mark price w.r.t current mark price after all trades have been opened. Close all positions in random order.
6	Intialize system parameters (vETH, vUSDC) Open 5 long and 10 short positions from 10 user accounts. Each account will open at least 1 short trade. Accounts can open long trades at random The collateral of short trades is 100 USDC and long trades is 200 USDC. Open trades in random order. The liquidity locked in long trades should be equal to the short trades. Update Mark Price to be greater than the current mark price. Close all positions in any order. Observe if the system can handle trade closures when the mark price is updated and the liquidity locked in long and short trades is the same but the no. of trades isn't. Repeat the experiment by reducing the mark price w.r.t current mark price after all trades have been opened. Close all positions in random order.

Table 5.2 continued from previous page

No.	Experiment Description
7	Intialize system parameters (vETH, vUSDC) Open 5 long and 5 short trades from any number of user accounts in any order. Choose the collateral of each trade at random and ensure the liquidity locked in long trades is greater than 10% of the liquidity locked in short trades. Note the mark price after all trades are open Update Mark Price to be greater than the current mark price. Close all positions in random order Observe if the system can handle trade closures when the mark price is updated and the liquidity locked in long and short trades is not the same. Repeat the experiment by reducing the mark price w.r.t current mark price after all trades have been opened. Close all positions in random order.
8	Repeat experiment 7 and ensure that the liquidity locked in short trades is greater than atleast 10% of the liquidity locked in long trades.
9	Perform experiments 7 and 8 and ensure that the difference in the liquidity locked in long and short trades is less than 10%.
10	Intialize system parameters (vETH, vUSDC) Open unequal number of long and short trades with multiple user accounts. Perform trades with high, medium and low collateral values. Ensure that the difference in liquidity locked in long and short trades is greater than 20% Note the mark price after all trades are open Update Mark Price to be greater than the current mark price. Close all positions in random order Observe if the system can handle trade closures when the mark price is updated and the liquidity locked in long and short trades is not the same. Repeat the experiment by reducing the mark price w.r.t current mark price after all trades have been opened. Close all positions in random order.

5.2 Results

This section provides the results of the experiments performed to evaluate the implemented DvAMM based perpetual swap exchange.

The results from performing the experiments seen in table 5.1 can be found in table 5.3.e The experiment numbers seen in table 5.3 corresponds to the experiment numbers seen in table 5.1.

The results generated from performing experiments mentioned in table 5.2 can be found in 5.4. The experiment numbers seen in table 5.4 corresponds to the experiment numbers seen in table 5.2.
Apart from the results obtained in 5.4, there are 2 common results attained on performing the experiments seen in table 5.2 for all 3 sets of initial parameter values. The common results attained are as follows:

1. It is observed that opening a position with the same collateral for all three sets of initial parameters seen in table 5.4 causes the mark price to diverge differently for each set. It is observed in all experiments that opening a long/short trade with the same collateral for all 3 sets of parameters causes the mark price to diverge more when the value of K is low compared to when the value of K is high.

2. It is observed that the slippage experienced by the exchange is highest when the system is initialised with set 1's parameter values (lowest K, vETH and vUSDC values). When the system is initialised with set 3's parameter values (highest K, vETH and vUSDC values), the slippage experienced is the lowest when opening and closing multiple trades in parallel.
   Slippage is the difference between the expected price of an asset when making a trade and the price at which the trade is actually executed (37). On performing the experiments seen in 5.2, it is observed that the size of K and virtual liquidity pool reserves are inversely proportional to the levels of slippage observed.

Table 5.3: Results of experiments performed to test if the implemented system can accurately perform functionalities of a vAMM based perpetual exchange.

Experiment No.	Result Obtained from experiments	Comparing results obtained with expected results
Initial Parameters: vUSDC = 100,000, vETH = 1000, Mark Price 100 USDC, K = 100,000,000		

Table 5.3 continued from previous page

Experiment No.	Result Obtained from experiments	Comparing results obtained with expected results
Experiments 1.1 and 1.2	Long trades open seamlessly. The vault balance is updated as each trade is opened. The dynamic vAMM calculations and exchange rates are formulaically correct. The calculations are verified by performing them by hand All trades close seamlessly. The system does not prevent the liquidation of any trades due to low funds in the vault. The vault balance is 0 after all trades have been closed	The results obtained by performing experiments 1.1 and 1.2 match the expected results of these experiments seen in table 5.1
Experiments 2.1 and 2.2	Short trades open seamlessly. The vault balance is updated correctly as each trade is opened. The dynamic vAMM token reserve calculations are correct when a long or short contract is opened. The calculations are verified by using Microsoft(MS) excel. The formula used by a constant product market maker (CPMM) is entered into excel and the calculation for each trade is performed and compared simultaneously on excel and the implemented system All trades close seamlessly, and the system does not prevent the liquidation of any trades due to low funds in the vault. The Vault balance is 0 after all trades are closed.	The results obtained by performing experiments 2.1 and 2.2 match the expected results of these experiments seen in table 5.1

Table 5.3 continued from previous page

Experiment No.	Result Obtained from experiments	Comparing results obtained with expected results
Experiment 3	All contracts open seamlessly. The vault is updated correctly as each trade is opened. The dynamic vAMM token reserve calculations are correct when a long or short contract is opened. The calculations are verified by using(MS) excel. The formula used by a constant product market maker (CPMM) is entered into excel and the calculation for each trade is performed and compared simultaneously on excel and the implemented system All trades close seamlessly. The system does not prevent the liquidation of any trades due to low funds in the vault. The Vault balance is 0 after all trades are closed.	The results obtained by performing experiment 3 matches the expected results of this experiments seen in table 5.1
Experiment 4.1	All contracts open seamlessly. The vault is updated correctly as each trade is opened. The dynamic vAMM token reserve calculations are correct when a long or short contract is opened. The calculations are verified by using excel. All trades close seamlessly. All short trades experience a loss. The value returned on closing a short trade is less than the deposited collateral on the trade All long trades experience a profit. The value returned on closing a long trade is more than the deposited collateral on the trade All trades close seamlessly. The system does not prevent the liquidation of any trades due to low funds in the vault. The Vault balance is 0 after all trades are closed.	The results obtained by performing experiment 4.1 matches the expected results of this experiment seen in table 5.1

Table 5.3 continued from previous page

Experiment No.	Result Obtained from experiments	Comparing results obtained with expected results
Experiment 4.2	Similar results to experiment 4.1 are obtained. The difference in results is that short traders do not experience a loss. The amount returned on closing a short trade is never less than the collateral deposited for that short trade	The results obtained by performing experiment 4.2 matches the expected results of this experiment seen in table 5.1
Experiment 5.1	All contracts open seamlessly. The vault is updated correctly as each trade is opened. The dynamic vAMM token reserve calculations are correct when a long or short contract is opened. The calculations are verified by using excel. All trades close seamlessly. All long trades experience a loss. The value returned on closing a long trade is less than the deposited collateral on the trade All short trades experience a profit. The value returned on closing a short trade is more than the deposited collateral on the trad The system does not prevent the liquidation of any trades due to low funds in the vault. The Vault balance is 0 after all trades are closed.	The results obtained by performing experiment 5.1 matches the expected results of this experiment seen in table 5.1
Experiment 5.2	Similar results to experiment 5.1 are obtained. The difference in results is that long traders do not experience a loss. The amount returned on closing a long trade is never less than the collateral deposited for that long trade	The results obtained by performing experiment 5.2 matches the expected results of this experiment seen in table 5.1

Table 5.4: Results of experiments performed to evaluate the behaviour of the DvAMM based system when the mark price is updated

Experiment No.	Experiment Results and Observations
Initial Parameters: Mark Price = 100 USDC/ETH Set 1:. vUSDC = 1000 , vETH = 10, K = 10,000. Set 2: vUSDC = 100,000 , vETH = 1000, K = 100,000,000. Set 3:. vUSDC = 10,000,000 , vETH = 100,000 , K = 1,000,000,000,000.	
Experiment 1	The system cannot close the long position for all 3 sets of initial parameters. The vault does not have enough balance to pay out the collateral and profit that would be received from closing the position
Experiment 2	The system cannot close the short position for all 3 sets of initial parameters. The vault does not have enough balance to pay out the collateral and profit that would be received from closing short the position
Experiment 3	The system liquidated and close out the long position for all 3 sets of initial parameters. The trader receives a loss on trade closure since the price of ETH in USDC fell after the long trade was opened. The vault has some additional balance left after the trade is closed.
Experiment 4	The system liquidated and closed out the short position for all 3 sets of initial parameters. The trader receives a loss on trade closure since the price of ETH in USDC rose after the short trade was opened. The vault has some additional balance left after the trade is closed
Experiment 5	The system successfully opened all trades and had 2000 USDC in it's vault after all trades were opened. The mark price was equal to the inital mark price of 100 USDC/ETH. The DvAMM based system successfully closed all positions. Updating the mark price altered the profit and loss margins recieved on the trades. The balance of the vault is 0 after all trades have closed.

Table 5.4 continued from previous page

Experiment No.	Experiment Results and Observations
Experiment 6	The system successfully opened all trades and had 2000 USDC in its vault after all trades had opened. The system is able to eventually close out all the trades when the mark price is updated to be higher than its current value after opening all trades. The system is also able to eventually close out all the trades when the mark price is updated to be lower than its current value after opening all trades. The trades had to be closed in a certain order for all trades to be liquidated. There were cases experienced during trade closure where one large profitable trade could not be closed until some smaller or loss-making trades were closed. After closing all trades in a certain order, the balance of the vault is 0 The results obtained for all 3 sets of initial parameter values are identical with the only difference being the profit and loss margins.
Experiment 7	The system fails to close out or liquidate trades after closing out a few trades. The implemented system always experiences a scenario where there are some trades left that cannot be closed out. The system is left in a state more trades cannot be closed until newer trades are opened that add liquidity to the vault. Even then, the system always remains in a state where all open trades can never be closed. The vault does not have enough balance to pay out the collateral and profit or loss that would be received from closing the remaining open positions. On repeating this experiment by updating the mark price to be lower than its current value after opening all trades, the system is observed to eventually close out all open positions. However, the system does experience cases where it cannot close out certain trades until more trades are opened or closed. The same results are obtained on performing experiment 7 for all 3 sets of initial parameter values

Table 5.4 continued from previous page

Experiment No.	Experiment Results and Observations
Experiment 8	On performing experiment 8 and updating the mark price to be lower than its current value after opening all trades, it is observed that the system always experiences a scenario where there are some trades left that cannot be closed out or liquidated. The vault does not have enough balance to pay out the collateral and profit or loss that would be received from closing the remaining open positions. On repeating this experiment by updating the mark price to be higher than its current value after opening all trades, the system is observed to eventually close out all open positions. However, the system does experience situations where it cannot close out certain trades until more trades are opened or closed. The system behaviour observed on performing experiment 8 for all 3 sets of initial parameter values is the same. However, the first set of initial parameter values experiences higher slippage for smaller collateral values (for e.g, a trade with collateral =500USDC)
Experiment 9	The system behaviour observed in performing experiment 9 is the same as the system behaviour experienced in experiments 7 and 8. There is a difference in results between experiments 7,8 and 9. It is observed that by reducing the difference in liquidity between long and shorts below 10% increases the total volume of trades (in USDC) that can be closed before the system cannot close more trades when the mark price of the system is updated such that it profits positions that have more liquidity locked in compared to the other position. When the mark price update profits the position with lesser liquidity, it is observed that all trades close eventually. There number of trades blocked from being closed are lesser as well. The same system behaviour is observed for all 3 sets of initial parameters

Table 5.4 continued from previous page

Experiment No.	Experiment Results and Observations
Experiment 10	On performing experiment 10 and updating the mark price to be either lower or higher than its current value after opening all trades, it is observed that the system mostly reaches a state where there are some trades left that cannot be closed out or liquidated. When all open trades are closed in a random order, the system always reaches a state where some trades cannot be closed due to a lack of funds in the vault. The system is able to close out all trades eventually when the liquidity locked in long and short trades is the same. However, when closing trades randomly under equal liquidity conditions, there are some open trades that cannot be closed out but others can. Trades with a high collateral value introduced higher slippage in the system for sets 1 and 2 with respect to set 3. The system behaviour observed for all 3 initial parameter sets are the same.

5.3 Discussion

This section discusses the results obtained in section 5.2 and evaluates the system based on its strengths, weaknesses, and limitations. This section further suggests solutions to address the limitations of the implemented system.

Section 5.3.1 discusses the system's performance when the mark price of the system is left unchanged. Section 5.3.2 details the impact of the K value on the system's performance. Section 5.3.3 evaluates the system's performance when the mark price of the system is configured. Section 5.3.4 addresses the limitations and weaknesses of the system and suggests solutions to tackle the system's limitations. Section 5.3.5 discusses the strengths of the implemented system and section 5.3.6 summarises the results discussed in this section.

5.3.1 System Performance when system's token reserves remain unchanged

Dynamic vAMM based perpetual exchanges need to correctly perform all the operations of a fully functional vAMM based perpetual exchange. Experiments seen in table 5.1 are performed to verify that the implemented dynamic vAMM based perpetual swap exchange

can operate as a fully functional vAMM based DEX.

The set of experiments seen in table 5.1 are performed with the virtual liquidity pool reserves and mark price of ETH/USDC set at the start of the experiment. The token reserves and mark price of the implemented system are never changed between or after any experiment. From the results obtained(see table 5.3), it is observed that the DvAMM always updates the virtual liquidity pool reserves accurately. The results seen in table 5.3 demonstrate that the vault always stores the correct amount of collateral when a trade is opened or closed. From the results seen in table 5.3, it is further observed that the implemented system is always able to liquidate trades and return the profits and losses along with the trade collateral to a user. There is never a scenario where the DvAMM based system cannot pay out a trader and close an open position when the token reserves of the system are not changed. It is also observed that the system never charges a trader more than the collateral a trader wishes to deposit for a trade.

The results obtained in table 5.3 verify that the implemented system can perform all operations of a virtual automated market-making protocol correctly. The results obtained in table 5.3 from performing experiments seen in table 5.1 verify that the dynamic vAMM based protocol can handle edge cases and normal scenarios that a vAMM based exchange would face in a production environment.

5.3.2 Impact of K value on the system's performance

This section discusses the importance of the K value in the implemented constant product market maker based DvAMM exchange.

Each experiment seen in table 5.2 is performed 3 times with different initial virtual token reserves and K values each time the experiment is repeated. This section discusses the difference in results obtained from performing these experiments with different initial parameter values. The results of performing the experiments seen in table 5.2 can be seen in section 5.2.

From the results seen in section 5.2 it is observed that the slippage experienced by traders on the system is inversely proportional to the value of the constant K. For an opening with a particular collateral, a dynamic vAMM with a lower K value experiences higher slippage, whereas a dynamic vAMM with a high K value experiences lower slippage.

Slippage is caused by three factors, volatility, trade size, and lack of liquidity (37).

Market volatility for a perpetual token pair and the trade size cannot be controlled by the system implemented. However, the lack of liquidity can be easily mitigated by controlling liquidity. Liquidity is the ease with which an asset can be bought or sold without affecting its price (37).

Since the implemented system has a virtual liquidity pool for its assets (ETH), the system can be initialized with a high value of virtual tokens (vETH) for its assets. Slippage can be reduced by increasing the amount of vETH, vUSDC and consequentially K. With a high amount of vETH tokens, the ease of buying ETH without affecting its price increases.

However, setting the value of K and virtual liquidity pool tokens has consequences. If the mark and spot price divergence is high and K is too high, arbitrageurs will not have adequate funds to keep the DvAMM mark price in line with the spot price (6).

From the results seen in section 5.2 it is also observed that the mark price divergence for a particular trade size is inversely proportional to the value of the constant K. The following example will provide a better understanding of the results obtained regarding mark price divergence. Suppose the current mark price is 1 ETH = 500 USDC. When a long trade with collateral = 500 USDC is opened, the following mark price divergence rates are attained for all 3 sets of initial parameters seen in table 5.4

1. Set 1, K=1000: The mark price changes from 100 to 225 USDC/ETH on opening the trade. A 125 % change in the mark price is observed.
2. Set 2, K = 100,000: The mark price changes from 100 to 101.0025 USDC/ETH on opening the trade. A 1.01% change in the mark price is observed.
3. Set 3, K = 10,000,000: The mark price changes from 100 to 100.01 USDC/ETH on opening the trade. A 0.01% change in the mark price is observed.

The mark price divergence rate can be controlled by setting the value of K high. However, as mentioned above, setting K too high will have a negative impact on arbitraging.
From the results and discussion regarding K values and virtual token reserves, it is important to understand that the K value initialized determines the slippage levels experienced by traders and arbitraging incentives. If K is low, the slippage and mark price divergence is high. If K is high, slippage and the mark price divergence for bigger trades is low; however, the arbitrageurs and arbitraging incentives are affected.

More research is needed to determine the ideal token reserves and K values for a vAMM based exchange. The K value needs to be set and optimized such that it balances slippage and arbitraging depending on the trading volume on the exchange. The implemented dynamic vAMM can have functionality added such that it dynamically scales K depending on the trading volume on the exchange to balance out slippage and arbitraging. Dynamic K scaling was not implemented since it requires data and analysis of how users would use the system in production. Using user data and trading volume information, a mechanism can be implemented to choose an optimal K value and then use a dynamic vAMM to scale the K value accordingly.

5.3.3 System Performance when the system's virtual token reserves change

Dynamic vAMMs are created to make existing vAMM solutions more adaptable to present external market conditions. This study implements a dynamic vAMM based perpetual swap exchange to analyse the working of a dynamic vAMM. The implemented perpetual swap exchange implements the core functionality of a dynamic vAMM which allows it to scale its virtual liquidity pool token reserves to adapt to external market conditions.

The experiments seen in table 5.2 test the functionality of the implemented DvAMM protocol when the virtual liquidity pool token reserves are updated during the normal functioning of the protocol. These experiments are performed to evaluate the implemented protocol's functionality and explore the shortcomings of the current implementation and dynamic vAMMs overall.

The scaling of the mark price in all experiments is performed to observe the behaviour of the system in scenarios where the mark price of the token pair (ETH/USDC) is not equal to the spot price, and the mark price is updated to be equal to the spot price to adapt to the external market conditions for the token pair. The results of these experiments demonstrate the impact of updating the mark price on the process of closing trades, opening existing trades and the users.

The results of the experiments performed on the implemented system can be seen in table 5.4. From the results, it is observed that there are various scenarios where the system cannot close out perpetual contracts when a trader requests to close their open position, after the mark price of the implemented system is updated.

From the results of experiment 5 seen in table 5.4, it is observed that the implemented system is able to close out all open trades when the liquidity locked in both positions is equal, the number of long and short contracts are equal, and the collateral deposited in each long and short trade are equal.

The results of experiments 1 and 2 show that the implemented DvAMM protocol cannot handle closing out open trader positions when there is only one type of position open, and updating the mark price benefits this type of position. In such cases, the vault does not have sufficient funds to pay out all traders. If all positions need to be closed out, the vault will need additional funds or a mechanism to pump money or tokens into it.

The results of experiments 3 and 4 seen in table 5.4 show that the implemented DvAMM protocol can smoothly handle closing out open trader positions when there is only one type of position open, and updating the mark price does not benefit this type of position. In such cases, the vault will have sufficient funds to pay out all traders and have some remaining

balance after all trades are closed out.

From the results of experiment 6 it is observed that when the liquidity in long and short positions on the system is equal, the implemented system is able to close out all long and short open positions eventually. The system does not experience a situation where, if there are 2 positions open, at least one of them can be closed. However, the DvAMM protocol does experience situations where some open trades cannot be closed, and the traders cannot be liquidated since the vault does not have enough funds to close these positions. These trades are dependent on other trades closing first such that these blocked trades can be closed. The system will eventually allow all trades to close out, but some unwanted dependencies between open trades are introduced when all open trades are being closed.
If a new trade on the system is opened, it may enable the blocked trade to be closed since the new trade added liquidity to the vault. However, opening new trades does not solve the issue of several contracts being dependent on other contracts to be closed so that they can be closed.

The results of experiments 7, 8, 9 and 10 can be seen in table 5.4 . In experiments 7, 8, 9 and 10, the liquidity locked in long and short positions is imbalanced. In such experiments, when the mark price is updated to profit the position with higher liquidity, it is observed that the implemented DvAMM based exchange cannot close any more open contracts after closing out a few open contracts. From the results of experiments 7, 8, 9 and 10, it is observed that the system always reaches a state where several remaining open contracts can never be closed.
The only case where the implemented DvAMM protocol will be able to close out all open contracts is when external funds are added to the vault. Liquidity provided to the vault from opening new trades does not count as external funds.
Despite the number of trades being equal in experiments 7, 8 and 9, the implemented DvAMM protocol cannot close out all trades. This observation from experiments 7, 8 and 9's results indicates that the liquidity locked in both positions is a better indicator than the number of trades to determine if there will be scenarios where the implemented system will not be able to close contracts after a certain point.

From the results of experiments 3, 4, 7, 8, 9 and 10, when the mark price is updated such that it introduces losses to the position with higher liquidity, it is observed that the vault has some funds remaining in it after all trades have been closed. This is because all trades in the exchange collectively experience a net loss, i.e., the positions with higher liquidity experience a higher loss, and the vault needs to return lesser USDC to these trades. Therefore, the vault has some remaining funds when all trades.

The experiments performed in table 5.2 evaluate if the implemented system can perform as a fully operational perpetual swap exchange when the implemented protocol is made to

adjust its mark price to adapt to external market conditions. The implemented system is not considered fully operational when the system cannot close one or more open contracts due to insufficient funds in the vault.

From the discussion of the results of these experiments above, this study confirms that the implemented DvAMM exchange cannot function as a fully operation perpetual swap exchange when the DvAMM protocol is made to update its mark price.

There is only one scenario experienced where the implemented DvAMM exchange is fully operational, and the system can adjust its mark price to the spot price without facing any problems. This scenario is experienced when the liquidity locked in both positions is equal, the number of long and short contracts is equal, and for each long trade with a certain collateral value, there is a short trade with the same collateral deposited. For all other scenarios, the system faces problems where it is blocked from closing contracts or can only eventually close out all open contracts since some contracts are dependent on others to be closed first.

This study observes that the implemented DvAMM protocol is locked from closing existing open contracts when there is an imbalance in the liquidity locked in long and short perpetual contracts on the exchange. The implemented exchange reaches a point where the vault does not have enough funds to close one or more remaining open contracts. The system experiences scenarios where the long-short imbalance and mark price changes will still allow the system to close out all contracts eventually. However, in eventual contract closure, some contracts cannot be closed until other contracts are closed first.

This study finds that when the mark price of the implemented protocol is changed, the imbalance between the liquidity locked in long and short positions is inversely proportional to the value of open positions that can be liquidated. If the imbalance in liquidity between both positions is low, the implemented protocol can liquidate more trades in terms of their monetary value. If the imbalance between both positions is high, the implemented protocol will liquidate lesser trades in terms of their monetary value.
This observation only holds true when at least 1 long and 1 short position is open.

Similarly, from the discussion of the results above, it is observed that when the imbalance in liquidity between both positions is low, the implemented system has a higher chance of eventually closing out and liquidating all open positions and the dependencies between trade closures are low. Conversely, when the imbalance in liquidity between both positions is high, the implemented system has a lower chance of eventually closing out and liquidating all open positions, and the dependencies between trade closures are high.
This observation only holds true when at least 1 long and 1 short position is open.

If the implemented perpetual swap exchange has liquidity locked in only one type of position,

the implemented exchange will either be able to close out all positions or the exchange will be locked from closing after closing a few. The implemented exchange will be able to close out all positions when the changing the mark price causes it to move in the opposite direction the traders had bet on (see results of experiments 3 and 4 in table 5.4 and their discussion above). The implemented exchange will not be able to close out several open positions when changing the mark price causes it to move in the direction the traders had bet on (see results of experiments 1 and 2 in table 5.4 and their discussion above).

The implemented DvAMM based perpetual swap exchange faces problems with contract closure when the mark price of the token pair (ETH/USDC) implemented is updated. Situations where the implemented exchange cannot close open contracts are not healthy for the traders and the implemented exchange. Users of the exchange will get frustrated with the implemented system if they cannot close their positions. Users who are earning profits or are experiencing a loss would like to exit their position, and being unable to do so locks their liquidity forcibly.
Situations where the implemented exchange cannot close open are problematic for the exchange as traders will keep losing more money than they put into the protocol. In a vAMM based exchange, user collaterals provide the exchange liquidity to facilitate trading. One traders' loss pays out the profits of another. Suppose one trader faces losses higher than their collateral, another trader or a set of traders would profit from these losses. The exchange would have to pay out the profits of the trader earning profits since one trader has experiences losses greater than the money they put into the protocol and cannot be liquidated.

5.3.3.1 Summarizing the system's performance when mark price of ETH/USDC is changed

On evaluating the implemented DvAMM exchange based on its ability to function correctly when the exchange's mark price is configured, it is found that the implemented exchange succeeds to operate as a fully functional exchange when 3 conditions are met. These conditions are:

1. The total liquidity locked in long and short open perpetual contracts is equal.
2. The number of long and short open perpetual contracts is equal.
3. An open long perpetual contract with a specific amount of deposited collateral is always matched with an open short perpetual contract with an equal amount of collateral deposited

However, the implemented DvAMM based perpetual swap exchange fails to operate as a fully functional perpetual swap exchange when all the 3 conditions above are not fulfilled,

and the mark price of the system is configured using the implemented DvAMM. In 7 out of 10 experiments seen in tables 5.2 and 5.4, the implemented system cannot close some or all open contracts due to a lack of funds in the vault. The implemented system locks the trades of a trader.

Overall the implemented DvAMM based perpetual exchange fails to operate as a fully functional perpetual swap exchange since the implemented DvAMM based exchange was not implemented such that the above 3 conditions are always true when a trade is opened.

If the conditions mentioned above are enforced on the implemented system, the DvAMM based exchange will function like an order-book model based exchange rather than an automated market-making protocol based exchange. Implementing the DvAMM system with the aforementioned conditions would be impractical and inefficient. Enforcing the conditions above would cause the system to lose all the advantages that an AMM based protocol provides. AMM protocols were invented to solve the issues of low liquidity faced by order-book models in decentralized exchanges. Therefore, enforcing conditions on the current system would not be a viable solution to solve the problems that render the current DvAMM implementation not fully operational.

For a dynamic vAMM exchange to be fully functional, the DvAMM exchange should allow new contracts to open and close at any point a trader desires. The dynamic vAMM exchange should be able to adapt its mark price of a token pair to the spot price and still be able to close out all open contracts accurately with the correct returns based on the new mark price.

To address the limitations and to make the implemented DvAMM based perpetual exchange fully operational, new solutions and mechanisms need to be designed and implemented. The limitations and weaknesses of the implemented system and their potential solutions are discussed in section 5.3.4

5.3.4 Addressing the limitations and weaknesses of the implemented system

This section discusses the limitations and weaknesses observed in the implemented dynamic vAMM based perpetual swap exchange and suggests solutions regarding the limitations of the implemented system.

The suggested solutions discussed in this section aim to provide insight to systems that aim to implement dynamic vAMMs in the future for their unique property of being able to adapt to external market conditions.
The suggested solutions discussed in the sections below provide insight into the challenges of implementing a dynamic vAMM protocol for scaling the mark price of a token pair, and

possible mechanisms to solve these challenges.

Section 5.3.4.1 suggests a mechanism that allows the protocol to collect additional funds to backup the vault so that the vault does not block trade closures when a token pair's mark price is updated.

From the discussion of results in section 5.3.3, it is observed that the implemented DvAMM system faces problems closing open contracts when the mark price of the ETH/USDC was updated. Section 5.3.4.2 suggests a checking mechanism that allows a DvAMM protocol to check if the vault and the suggested fee collection mechanism discussed in section 5.3.4.1 can support the update to the mark price of a token pair. Supporting an update to the mark price of a token pair means that the DvAMM protocol can handle closing all open perpetual contracts after the mark price of that token pair has been updated.

Based on the fee collection and checking mechanisms, section 5.3.4.3 discusses when should a system consider updating its mark price so that it can adapt to present market conditions. Section 5.3.4.4 details an algorithm to automate the process of configuring the mark price of a token pair based on all the proposed solutions in 5.3.4.1, 5.3.4.2 and 5.3.4.3.

5.3.4.1 Insurance funds through transaction fees or taker fees

The current implementation of the dynamic vAMM protocol cannot work as a fully functional perpetual swap exchange because scaling the mark price affects profits and losses of open trades. The profits and losses for each trader are withdrawn from the vault. Therefore, adjusting the mark price for a token pair can negatively impact the vault where the vault may not have enough funds to pay out all traders if all traders decide to close their perpetual contract.
To make the system fully functional and to address the limitations the system faces when its mark price and virtual liquidity pool token reserves have changed, a backup funding mechanism is suggested so that the vault can pay out traders when a trade is closed.

To ensure that the vault has funds to close open contracts, the system needs to have a mechanism to provide additional funds to the vault so that traders are not blocked from closing their open positions and trade can be liquidated. This mechanism can gather funds by collecting a small fee each time a trader opens a trade. The fee charged is a small percentage of the trade collateral and not the total value of the trade being opened. The fee charged by the implemented system can be called either a 'transaction fee' or a 'taker fee' (55) and can be stored in a smart contract similar to the vault.

The initial fee charged by the system will be set to 0.1% of the total collateral being deposited. The initial fee is set to 0.1% based the fee perpetual protocol (56) and OKX exchange (57) charges its users. The transaction or taker fee stored will act as insurance for

paying out traders whenever the mark price of a token pair needs to be adjusted to the spot price of that token pair.

Having a transaction fee reduces the incentive to trade on a perpetual exchange. To ensure that the implemented system has lower fees, it is important to consider the aspect of dynamic fee percentages. If the trading volume on the exchange is high and the amount of fees stored by the exchange is high, the fee percent charged should be reduced. Inversely, if the number the trading volume on the exchange is low and the fee stored by the exchange is low, the fee percentage should be the highest, i.e., 0.1

5.3.4.2 A mechanism to check the feasibility of updating the system's mark price

Implementing the concept of charging a dynamic fee each time a contract is opened will contribute to an "insurance fund" to back up the vault. However, this insurance fund will not guarantee that the system will have sufficient funds to pay out all traders if they decide to close their open perpetual contract after the mark price has been adjusted. Therefore, this study suggests an additional checking mechanism to inspect if the DvAMM protocol can update its virtual liquidity pool reserves and mark price such that the update does not prevent the closing of any open contracts.

Before the mark price of a token pair in a DvAMM protocol is adjusted to match the spot price, the DvAMM protocol needs to know if the vault and fees collected can handle closing out all open contracts once the update to the mark price is performed.
Hence, a checking mechanism needs to be implemented to check if the total combined funds from the collected fees and vault would be able to handle closing out all open trades with the potential new mark price. The protocol will need to discover and calculate two key values to implement such a checking mechanism. These values are:

1. The total amount of funds stored in the vault and the total amount collected as fees.
2. The combined total amount of funds that will have to be returned to the users by the dynamic vAMM protocol if all current open contracts are closed.
   This total return amount will be calculated based on the new value of the mark price that the system wants to update to.
   Performing this calculation will provide the DvAMM protocol with an idea of how much money the system needs to have before it can update the mark price

Example 5.3.4.2 below demonstrates why the values mentioned above are important to discover. Example 5.3.4.2 below also demonstrates a checking mechanism that will use the values mentioned above to discover if the mark price update will let contracts close or not.

Example 5.3.4.2:
Consider the following market conditions in a DvAMM based perpetual exchange for ETH/USDC:

- Mark price: 1 ETH = 100 USDC.
- Spot Price: 1 ETH = 150 USDC
- The current vault balance can fund the closing of all open trades

Suppose the system wants to update its mark price to the spot price to adapt to market conditions for ETH/USDC. Before updating the mark price, the system needs to check if the vault and fees collected can handle closing out all open contracts once the mark price is updated. The system can carry out the following steps to perform this check:

1. The system will first need to simulate and find the total amount that will need to be returnedto the traders if all open contracts are closed when the mark price is 1 ETH= 150 USDC and vETH, vUSDC token reserves reflect this mark price. The amount found can be called 'potentialContractReturnValue'.
2. To guarantee that the protocol can perform contract closures after mark price update, the funds stored in the 'vault + fees' must be greater than the 'potentialContractReturnValue'.
3. If the 'vault + fees' >= 'potentialContractReturnValue', the system can safely update the mark price to the spot price. If 'vault + fees' < 'potentialContractReturnValue', then updating the mark price will potentially lead to scenarios where the vault and fees combined will not be able to fund the closure of one or more open trades.

5.3.4.3 Parameters to determine when the mark price should be configured

The problems that cause the implemented DvAMM system not to be a fully functional configurational exchange can be solved by combing the solutions suggested in sections 5.3.4.1 and 5.3.4.2.

Assuming that the developed system implements the insurance fund mechanism and a checking mechanism to communicate to the exchange that the mark price can be updated to the spot price, it is important to understand when the mark price of the implemented system should be updated.

It is impractical to configure the mark price of a dynamic vAMM exchange whenever it is possible for the exchange to do so without locking users from closing trades. If a dynamic vAMM is constantly configured, it will deplete the exchange's insurance fund. If the fee pool is depleted, the DvAMM based exchange will not be able to configure the mark price when the functionality is needed more. For example, it is more important for a system to be able

to configure its mark price when the mark price has diverged more than 15% from the spot price (mark-spot spread $> 15\%$) compared to when the mark-spot price divergence is only 5% mark-spot spread $< 5\%$).

The purpose of having a configurable or dynamic vAMM is to ensure that an exchange can adapt its mark price to external market conditions and is more flexible to volatile price changes. When the mark-spot spread of a token pair in a perpetual exchange is low, the difference between the mark and spot prices can be reduced through funding payments and arbitrageurs.

Implementing a dynamic vAMM benefits an exchange when the mark-spot spread of a perpetual exchange is high. Cryptocurrencies such as ETH and BTC are volatile assets (58) (59) and the prices of these cryptocurrencies changes sharply based on market sentiment (59). The mark-spot spread of an ETH/USDC or BTC/USDT perpetual contract can drastically increase at any point in time due to the volatility of ETH and BTC. A dynamic vAMM is valuable and should be configured to reduce the mark-spot spread in high mark-spot price divergence scenarios. For the dynamic vAMM to be ready for mark price configuration changes, the fee pool should not be depleted and should have sufficient funds. Suppose a dynamic vAMM is constantly configured to balance our small mark-spot spreads. In that case, the exchange's insurance fund will deplete quicker, and the dynamic vAMM will be rendered unconfigurable in periods of high volatility.

Drift protocol's preliminary analysis indicated that a good time to adjust the mark price closer to the spot price is when the mark-spot spread is less than 10%, and the imbalance between long and short traders is high resulting in high funding rates (38).
Hence a good starting point for an exchange implementing a safe and fully functional DvAMM would be to follow Drift protocols conventions on when to configure the mark price of a token pair.

Every perpetual contract is unique, and the price volatility profile of each token or cryptocurrency is different. Therefore, the metrics to determine when it is a good time to configure the mark price of a token pair to be closer to the spot price would depend on how volatile a token is itself. An exchange would have to analyse the volatility profile and trading patterns of a perpetual contract on the exchange before deciding the best metrics to configure the mark price of that contract. An exchange would also have to analyse the volume of liquidity, arbitraging rate, and fund rates for a perpetual token pair to understand when the best time would be to configure the mark price for this perpetual token pair.

In summary, various parameters such as the mark-spot spread, long-short imbalance, funding rates, token pair volatility and token pair perpetual trading demand need to be analysed to determine when an exchange should configure the mark price of a token pair. Intensive research for each perpetual token pair would need to be performed to determine the best

values of these parameters, such that configuring the mark price of a token pair benefits an exchange.

5.3.4.4 A potential solution to automating mark price configurations

The current implementation of a DvAMM based exchange provides a semi-manual mechanism to configure the mark price of a token pair. The implemented mechanism to configure the mark price is semi-manual as the researcher only needs to click a button on the UI to configure the mark price and does not need to manually change the token reserves to update the mark price to a desired value.

Semi-manually configuring the mark price of a token pair on an exchange introduces additional risks in the form of human errors. Therefore, to mitigate risks associated with semi-manually configuring the mark price of an exchange, an automated mechanism to configure the mark price of a token pair should be designed, developed, and tested.

Let us assume that the limitations of the implemented DvAMM based exchange are mitigated by the solutions suggested in sections 5.3.4.1 and 5.3.4.2, and the system knows the necessary parameters to determine when the mark price of the ETH/USDC token pair should be configured. Based on these assumptions, this study suggests a potential algorithm to automate the process of configuring the mark price of the implemented system. The suggested algorithm is detailed below:

Step 1: Check system parameters that indicate when the mark price should be configured, to determine if the mark price of ETH/USDC should be updated to its spot price. For example, check if ETH/USDC mark-spot spread>10% and long-short imbalance > 50%.

Step 2: If system parameters indicate that the mark price should be updated to the spot price, perform the following steps:

(a) Using the checking mechanism (see section 5.3.4.2) check if the total combined funds (collected fees + vault) are sufficient to handle closing out all open trades if the mark price is updated to the spot price.

i. If the total combined funds are sufficient to handle closing out all open trades, update the mark price to be equal to the spot price using the DvAMM. Jump to step 3

ii. If the total combined funds are not sufficient to handle closing out all open trades the spot price, find a value closest to the spot price that the mark price can be updated to and store this value in a variable 'closestPossibleMarkPrice'. Inform the system admins that the mark price cannot be updated to be equal to spot price and the closes spot price

configuration is the value stored in 'closestPossibleMarkPrice'. Jump to step 3.

- Note: The system admins can then inform this algorithm or another system to configure mark price = 'closestPossibleMarkPrice' or not.

Step 3: Perform step 1 after waiting for a certain time interval.

5.3.5 Strengths of the implemented DvAMM based system

This sections, discusses the strengths of the system that are observed based on the experiments performed on the system and their results.

A strength of DvAMM based exchange is its ability to allow the values of K and the virtual liquidity pool token reserved to be scaled manually to reduce the amount of slippage and increase the volume of assets in the pool. Increasing the number of assets in the liquidity pool increases the ease of trading an asset without affecting its price and reduces the slippage faced by traders. Another strength of the system is that it provides the potential for dynamically scaling K. The current system implementation scales the value of K to adjust the mark prices. However, a mechanism to scale the token reserves and K value can be formulated based on the current implementation. The implemented system provides insights into how future projects may look to configure K values for balancing slippage and arbitraging incentives. Dynamically scaling K is better than manually scaling K and virtual token reserves because manual scaling of system parameters, especially in production environments, will lead to increased operational and trading risks. A strength of the current implementation is that it opens the door to exploring mechanisms to scale K for various beneficial purposes.

Another strength of the implemented DvAMM based exchange is that the system explores a simple and easy to implement mechanism to allow the mark price of a perpetual contract token pair to be adjusted. The mechanism implemented to configure the mark price of tokens correctly adjusts the mark price of a token pair to the desired spot price without any errors. The implemented mechanism enables an exchange to adapt to market conditions for a token pair. It is true that the implemented system overall is not fully functional due to contract closure problems and requires additional mechanisms to render it fully functional. However, the system does implement a mechanism that is light, has fast execution times and is simple to understand.

5.3.6 Summarising the discussion of results

The discussion section first discussed the system's performance when the mark price was not configured in section 5.3.1. Section 5.3.1 discussed the results observed in table 5.3

presented in the Results section (section 5.2). From the current implementation, we found that the implemented DvAMM based exchange performed as a fully functional exchange when the mark price was not left unchanged. Section 5.3.2 discussed the impact of the K value on the system's performance.
Section 5.3.3 discussed the results observed in table 5.4 presented in the Results section (section 5.2). It was found that the implemented DvAMM based exchange failed to operate as a fully functional perpetual swap exchange when the mark price was configured.
Section 5.3.4 discusses the limitations of the system and discusses the reasons behind the system failing to operate as a fully operational perpetual swap exchange. Section 5.3.4 addressed the discussed limitations by suggesting possible solutions to solve the problems faced by the implemented system. The solutions provided in section 5.3.4 provide insight on how to develop a fully functional DvAMM based exchange that can configure its mark price without locking users from closing trades. Finally, section 5.3.5 discusses the strengths of the implemented DvAMM based exchange.

5.4 Critical Review of the Implemented System

This section critically reviews the implemented DvAMM based perpetual swap exchange based on the design and implementation of the system and results obtained by experimentation on the system.

A major shortcoming of the implemented system is that it does not implement liquidations. In the current implementation of the DvAMM based exchange, if a trader's total return amount for a perpetual contract goes below the collateral (i.e., collateral + profit/loss <= 0), the system does not automatically close out the perpetual contract. Since liquidations are not implemented as a part of the system, this research does not study the impact of liquidations on contracts when the mark price of a token pair is configured. If liquidations were implemented, this research would have observed the impact of mark price configurations on perpetual contract liquidations. Furthermore, if liquidations were implemented, this study would be able to suggest mechanisms or solutions regarding how a dynamic vAMM based exchange could handle liquidations when the mark price of a token pair is configured.

Another shortcoming of the implemented system is that the current implementation does not provide any mechanisms to simulate multiple users opening and closing perpetual contracts on the implemented system. Since each contract had to be manually opened and closed by the researcher, performing experiments was slower, and only a certain number of trades could be performed at a certain point of time from a limited number of accounts. Simulating multiple users trading on the implemented system would have enabled this research to perform a more diverse range of experiments. Simulating trades on the implemented system from multiple user accounts would also allow the DvAMM based

exchange to be exposed to and tested on various scenarios rather than just a few predetermined edge case scenarios. Therefore, simulating user trades would have helped this study evaluate the system based on a more diverse range of experiments.

A minor problem with the system's implementation is that the system does not implement and use ERC20 tokens but rather uses a locally built token for the purposes of testing. Implementing a local token removes the complexity of finding an external faucet to request funds from. With a local token and faucet, this research can study the impact of large trade sizes on the implemented system. However, implementing ERC20 tokens would have enabled the implemented system to create perpetual markets that are not just based on USDC. For example, a perpetual market for ETH/LINK (chainlink) or LINK/BAT (basic attention token) could have been created. Implementing multiple perpetual markets.

The system developed implements most of the functionalities of a perpetual swap exchange and focuses sharply on the functionality this study focuses on, which is the dynamic configuration of a vAMM, allowing a vAMM to adapt its state to external market conditions. Mitigating the above-mentioned shortcomings would have allowed the implemented system to be more production-ready. It would have allowed the system to be tested on multiple scenarios and multiple simultaneous users trading.

6 Conclusion

Decentralized derivative trading through automated market making is relatively new and in its early stages of growth. Despite AMMs and vAMMs being highly innovative solutions, they are not the most optimal solutions and are insensitive to external market conditions and the internal conditions of the system implementing them. Existing vAMM protocols manually scale the K value and virtual liquidity pool token reserves of a vAMM to balance the tradeoffs between slippage and arbitraging incentives (6). Manually scaling K leads to higher operational risks. Lower risk and improved mechanisms are needed to configure a vAMM so that vAMM based solutions are more optimal, flexible, and sensitive to external market conditions.

This research aimed to design, implement, test, and evaluate a perpetual swap exchange by implementing a mechanism that configures a vAMM to dynamically adjust its parameters so that the vAMM can adapt to external market conditions.
A mechanism to configure token reserves and the K value of a vAMM was designed and implemented on a perpetual swap exchange. The dynamic vAMM was built on top of the design of a stock vAMM. The configurable or dynamic vAMM enabled the implemented perpetual swap exchange to adjust its mark price for the ETH/USDC perpetual contract to its spot price. Having the functionality to adjust the mark price through a dynamic vAMM implementation allowed the implemented system to adapt to market volatility for the ETH/USDC token pair.

A set of experiments were then performed on the implemented DvAMM based perpetual swap exchange to evaluate the performance, limitations, and feasibility of the implemented dynamic vAMM.
It was found that the implemented dynamic vAMM can easily adjust its token volumes and K to adapt its mark price to any given spot price. However, the evaluation showed that the implemented dynamic vAMM failed to perform as a viable solution for perpetual swap exchanges because configuring the mark price of the implemented system would lock several perpetual contracts from being closed on the exchange. The reason for perpetual contracts being locked from being closed after configuring mark price was because of insufficient funds in the vault to support contract closures.

This study also found that the implemented system performed correctly when the DvAMM was never configured. The experiments performed on the system also showed that the DvAMM could be a viable solution for mark price adaption if some conditions were enforced on the system. However, it was found that enforcing these conditions on the exchange would turn into an order book model based exchange.

The reasons behind the failure of the implemented DvAMM as a viable and sustainable solution for perpetual swap exchanges were examined. Potential solutions were suggested to address the limitations of the vault having insufficient funds when the mark price is configured. It was discovered that having an insurance fund (formed by transaction or taker fee on each trade) backing the vault could potentially solve the issues of lack of funds in the system when the mark price is configured. Mechanisms to know when to configure the system's mark price and mechanisms to check if it is possible to update the system's mark price were proposed. Finally, an algorithm was proposed to automate the process of configuring the mark price of the implemented system.

The performance of the implemented system suggests that the dynamic vAMMs need additional mechanisms to be designed and implemented along with them so that a perpetual exchange can adapt to external market conditions.
The implemented DvAMM token reserve and K value adjustment mechanisms are not a complete solution for perpetual exchanges to adapt to market conditions. The dynamic vAMM needs additional mechanisms to support it to be a sustainable and fully functional solution. The supporting mechanisms include having an insurance fund, a mechanism to check if it is safe to configure the DvAMM and a mechanism to discover the best parameters to configure the mark price of a token pair. These mechanisms need to be explored and tested to develop a complete DvAMM solution.

The implemented dynamic vAMM is not a complete solution to address the problems of the inflexibility of vAMMs. To make DvAMMs a complete solution, additional mechanisms need to be designed, adding more complexity to the system. However, DvAMMs are a step forward towards exploring the DeFi perpetual swaps ecosphere and finding a solution to solve various existing problems in the automated market-making ecosystem.

6.1 Future Work

The exploration of dynamic vAMMs performed in this dissertation opens doors to multiple areas that need to be researched in the future to develop viable solutions that enable automated market-making protocols to adapt to market volatility. The proposed areas of future work and research are proposed below:

6.1.1 Different approaches to configuring vAMMs

The current research configures a vAMM to adapt a perpetual contract's token pair price to its mark price by scaling the virtual liquidity pool tokens and K values. However, this is only one method wherein a vAMM can be configured to adapt mark prices to spot prices. This research suggests that multiple methods to configure a vAMM should be explored in the future. Having multiple techniques to achieve the same goal will allow for the fabrication of more robust and viable solutions to dynamic automated market-making

6.1.2 Parameters to discover the best time to configure a vAMM

This research explored a potential mechanism for the system to know when its mark price should be updated to the spot price. Various parameters and values must be known so that a system or a system admin can be aware of when the mark price needs to be configured. Based on the study performed, this dissertation suggests that it is important for future studies to explore and find the most suitable parameters that would assist in determining the best scenarios to configure a vAMM.

6.1.3 A system or mechanism to check if a DvAMM based exchange can update its mark price

For a dynamic vAMM based system, it is important to know that updating the mark price of a token pair will not block trade closures. Methods to design and implement algorithms or mechanisms to perform such checks need to be researched. Configuring a vAMM would be safer and more secure if a robust and efficient mechanism could be developed to perform the checks mentioned above.

6.1.4 Impact of configuring a vAMM on trade liquidations

This research evaluated the performance of configuring the mark price of a token pair via a DvAMM on the implemented system. This research found that configuring the mark price can result in open trades being forcefully liquidated since the margin of these trades falls below the deposited collateral. This dissertation suggests that future projects should investigate approaches on how to handle liquidations caused by mark price updates because forced liquidations may lead to distress amongst the users of a dynamic vAMM based exchange.

6.1.5 Optimal K value

The current research discusses the impact of the K value on a vAMM based system. Adjusting or initializing the K value to be optimal would benefit AMM-based derivative

exchanges as it would help them balance out slippage rates and arbitraging incentives. This research suggests that future studies should explore parameters, methods, and algorithms that would help an AMM or vAMM based exchange find the most optimal K value for that exchange at a particular time. This dissertation suggests that future research projects should also find strategies and methods to dynamically scale the K value of an AMM or vAMM to balance the rate of slippage and arbitraging.

Bibliography

[1] AMM V2 Uniswap. How uniswap works: Uniswap. URL https://docs.uniswap.org/protocol/V2/concepts/protocol-overview/how-uniswap-works.

[2] Osmosis Labs. Constant product. URL https://osmosis.gitbook.io/o/basic-concepts/amm/constant-function#:~:text=Constant%20function%20means%20that%20no,functions%20are%20utilized%20as%20well.

[3] Jason Fernando. What is a derivative?, Feb 2022. URL https://www.investopedia.com/terms/d/derivative.asp#:~:text=Derivatives%20are%20financial%20contracts%2C%20set,fluctuations%20in%20the%20underlying%20asset.

[4] Etienne Royole. Defi pulse - the decentralized finance leaderboard: Stats, charts and guides: Defi pulse. URL https://www.defipulse.com/.

[5] CoinMarketCap. Top cryptocurrency decentralized exchanges ranked, Feb 2022. URL https://coinmarketcap.com/rankings/exchanges/dex/.

[6] Perpetual Protocol. A deep dive into our virtual amm (vamm), Jul 2021. URL https://blog.perp.fi/a-deep-dive-into-our-virtual-amm-vamm-40345c522eeb.

[7] GartnerInc. Definition of blockchain gartner information technology glossary. URL https://www.gartner.com/en/information-technology/glossary/blockchain.

[8] Blockchain as a data structure, 2018. URL https://academy.horizen.io/technology/expert/blockchain-as-a-data-structure/.

[9] Andreas M. Antonopoulos and Gavin Wood. *What is Ethereum?*, page 1–12. O'Reilly, first edition, 2019.

[10] Andreas M. Antonopoulos and Gavin Wood. *The Ethereum Virtual Machine*, page 297–318. O'Reilly, first edition, 2019.

[11] Andreas M. Antonopoulos and Gavin Wood. *Smart Contracts and Solidity*, page 127–160. O'Reilly, first edition, 2019.

[12] What are ethereum tokens? a guide to the asset types of defi, May 2021. URL https://blog.makerdao.com/what-are-ethereum-tokens-a-guide-to-the-asset-types-of-defi/.

[13] Joshua. Token standards, Oct 2021. URL https://ethereum.org/en/developers/docs/standards/tokens/#top.

[14] Token tracker | etherscan, Feb 2022. URL https://etherscan.io/tokens.

[15] Adam Hayes. How do futures contracts work?, Jan 2022. URL https://www.investopedia.com/terms/f/futurescontract.asp.

[16] Gordon Scott. Futures exchange definition, Jan 2022. URL https://www.investopedia.com/terms/f/futuresexchange.asp.

[17] CME Group. Definition of a futures contract - cme group, 2021. URL https://www.cmegroup.com/education/courses/introduction-to-futures/definition-of-a-futures-contract.html.

[18] CME Group. The power of leverage - cme group, 2021. URL https://www.cmegroup.com/education/courses/understanding-the-benefits-of-futures/the-power-of-leverage.html.

[19] CME Group. Margin: Know what's needed - cme group, 2021. URL https://www.cmegroup.com/education/courses/introduction-to-futures/margin-know-what-is-needed.html.

[20] James Chen. Margin call, Jan 2022. URL https://www.investopedia.com/terms/m/margincall.asp.

[21] Jake Frankenfield. Tether (usdt), Jan 2022. URL https://www.investopedia.com/terms/t/tether-usdt.asp.

[22] Coinbase. What is a stablecoin?, 2022. URL https://www.coinbase.com/learn/crypto-basics/what-is-a-stablecoin.

[23] funding fee. Introduction to binance futures funding rates, 2020. URL https://www.binance.com/en-NG/support/faq/360033525031/.

[24] Changelly. Funding fee, Aug 2021. URL https://support.changelly.com/en/support/solutions/articles/14000131772-funding-fee.

[25] James Chen. Spot price definition, May 2021. URL https://www.investopedia.com/terms/s/spotprice.asp.

[26] What is a dex? URL https://www.coinbase.com/learn/crypto-basics/what-is-a-dex.

[27] Binance Academy. Order book, Apr 2020. URL https://academy.binance.com/en/glossary/order-book.

[28] Andrey Sergeenkov. What is an automated market maker?, Aug 2021. URL https://www.coindesk.com/learn/2021/08/20/what-is-an-automated-market-maker/.

[29] Vitalik Buterin. R/ethereum - let's run on-chain decentralized exchanges the way we run prediction markets, 2017. URL https://www.reddit.com/r/ethereum/comments/55m04x/lets_run_onchain_decentralized_exchanges_the_way/.

[30] CoinMarketCap. Top cryptocurrency decentralized exchanges ranked, Feb 2022. URL https://coinmarketcap.com/rankings/exchanges/dex/.

[31] CoinMarketCap. Liquidity pool: Coinmarketcap, Aug 2021. URL https://coinmarketcap.com/alexandria/glossary/liquidity-pool.

[32] James Chen. Arbitrageur, May 2021. URL https://www.investopedia.com/terms/a/arbitrageur.asp.

[33] Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, AFT '20, page 80–91, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381390. doi: 10.1145/3419614.3423251. URL https://doi.org/10.1145/3419614.3423251.

[34] Dmitriy Berenzon. Constant function market makers: Defi's "zero to one" innovation, May 2020. URL https://medium.com/bollinger-investment-group/constant-function-market-makers-defis-zero-to-one-innovation-968f77022159.

[35] Binance Academy. Impermanent loss explained, Aug 2021. URL https://academy.binance.com/en/articles/impermanent-loss-explained.

[36] Alex. On vamm's unnecessity for a liquidity pool, May 2021. URL https://qmeasuregrey.medium.com/on-vamms-unnecessity-for-a-liquidity-pool-6662751f3acb.

[37] Perpetual Protocol. What is slippage?, Jan 2022. URL https://blog.perp.fi/what-is-slippage-5d5b5bcb418e.

[38] Damian Chen and Cindy. Drift's dynamic amm, Mar 2022. URL https://docs.drift.trade/drifts-dynamic-amm.

[39] Wei-Meng Lee. Using the metamask chrome extension. In *Beginning Ethereum Smart Contracts Programming*, pages 93–126. Springer, 2019.

[40] Perpetual Protocol. A deep dive into our virtual amm (vamm), Jul 2021. URL https://blog.perp.fi/a-deep-dive-into-our-virtual-amm-vamm-40345c522eeb.

[41] Sam Richards. Smart contract languages, 2022. URL https://ethereum.org/en/developers/docs/smart-contracts/languages/.

[42] Mudabbir Kaleem, Anastasia Mavridou, and Aron Laszka. Vyper: A security comparison with solidity based on common vulnerabilities. In *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 107–111. IEEE, 2020.

[43] Andreas M. Antonopoulos and Gavin Wood. *Smart Contracts Security*, page 171–220. O'Reilly, first edition, 2019.

[44] Gregory McFarland. The benefits of bottom-up design. *SIGSOFT Softw. Eng. Notes*, 11(5):43–51, oct 1986. ISSN 0163-5948. doi: 10.1145/382298.382368. URL https://doi-org.elib.tcd.ie/10.1145/382298.382368.

[45] Sayan Kar, Yash Kothari, and Nimish Agrawal. Figment learn: Create an automated market maker (amm) on avalanche, Oct 2021. URL https://learn.figment.io/tutorials/create-an-amm-on-avalanche#prerequisites.

[46] Sayan Kar. Sayankar/avalanche-amm: Learn how to make an automated market maker. this tutorial is published on figment.io. the link to the tutorial can be found on the website link., Oct 2021. URL https://github.com/SayanKar/avalanche-amm.

[47] reactjs.org. Getting started. URL https://reactjs.org/docs/getting-started.html.

[48] ethers. URL https://docs.ethers.io/v5/.

[49] Geoff Hayes. The beginners guide to using an ethereum test network, Jan 2019. URL https://medium.com/compound-finance/the-beginners-guide-to-using-an-ethereum-test-network-95bbbc85fc1d.

[50] Tom Hay. Ethereum javascript libraries: Web3.js vs ethers.js: An update, Nov 2021. URL https://blog.infura.io/ethereum-javascript-libraries-web3js-ethersjs-nov2021/.

[51] REMIX. Ethereum ide amp; community. URL https://remix-project.org/.

[52] Ganache. URL https://trufflesuite.com/ganache/.

[53] Chris Ward. URL https://docs.soliditylang.org/en/v0.8.10/types.html.

[54] OpenZeppelin. Safemath. URL https://docs.openzeppelin.com/contracts/2.x/api/math.

[55] Binance. Trade crypto futures: How much does it cost?, Jun 2021. URL https://www.binance.com/en/blog/futures/trade-crypto-futures-how-much-does-it-cost-4214998246849022 39.

[56] perp.fi. What do i need to start trading?, 2021. URL https://docs.perp.fi/faqs/trading-faq.

[57] Okx. Trading fee: Futures fee: Crypto exchange fees: Fee tiers: Options trading fees. URL https://www.okx.com/fees.html.

[58] Jaysing Bhosale and Sushil Mavale. Volatility of select crypto-currencies: A comparison of bitcoin, ethereum and litecoin. *Annu. Res. J. SCMS, Pune*, 6, 2018.

[59] Anne Haubo Dyhrberg. Bitcoin, gold and the dollar – a garch volatility analysis. *Finance Research Letters*, 16:85–92, 2016. ISSN 1544-6123. doi: https://doi.org/10.1016/j.frl.2015.10.008. URL https://www.sciencedirect.com/science/article/pii/S1544612315001038.