FEDERAL UNIVERSITY OF SANTA CATARINA
TECHNOLOGY CENTER
AUTOMATION AND SYSTEMS DEPARTMENT
UNDERGRADUATE COURSE IN CONTROL AND AUTOMATION ENGINEERING

Pedro Yves Fracari Ceripes

Dynamic fee algorithm for maximizing liquidity provider returns in Automated Market Makers

Florianópolis
2024

Pedro Yves Fracari Ceripes

Dynamic fee algorithm for maximizing liquidity provider returns in Automated Market Makers

Final report of the subject DAS5511 (Course Final Project) as a Concluding Dissertation of the Undergraduate Course in Control and Automation Engineering of the Federal University of Santa Catarina.
Supervisor: Prof. Alex Sandro Roschildt Pinto, Dr.
Co-supervisor: Fabio Simones Mendes, Eng.

Florianópolis
2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Ceripes, Pedro Yves Fracari
Dynamic fee algorithm for maximizing liquidity provider returns in Automated Market Makers / Pedro Yves Fracari Ceripes ; orientador, Alex Sandro Roschildt Pinto, coorientador, Fabio Simones Mendes, 2024.
69 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia de Controle e Automação, Florianópolis, 2024.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Dynamic Swap Fee. 3. Automatic Market Maker.. 4. Decentralized Finance. 5. Smart Contracts. I. Pinto, Alex Sandro Roschildt. II. Mendes, Fabio Simones. III. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. IV. Título.

Pedro Yves Fracari Ceripes

Dynamic fee algorithm for maximizing liquidity provider returns in Automated Market Makers

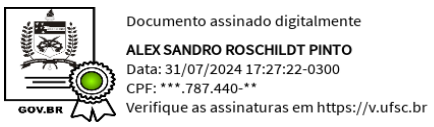
This dissertation was evaluated in the context of the subject DAS5511 (Course Final Project) and approved in its final form by the Undergraduate Course in Control and Automation Engineering

Florianópolis, June 10th, 2024.

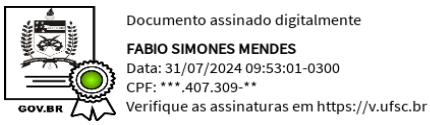
Prof. Marcelo De Lellis Costa de Oliveira, Dr.
Course Coordinator

Examining Board:

Prof. Alex Sandro Roschildt Pinto, Dr.
Advisor
UFSC/CTC/INE



Fabio Simones Mendes
Supervisor
Bleu Studio



Bruno Machado Pacheco, Msc.
Evaluator
UFSC/CTC/DAS

Prof. Hector Bessa Silveira, Dr.
Board President
UFSC/CTC/DAS

ACKNOWLEDGEMENTS

This work is first dedicated to my beloved family. Moving to another city was not easy for me and my parents, Beatriz and Edvaldo, but your support was essential during this time. Throughout my educational journey, you always pushed me to achieve my full potential, for which I am deeply grateful. To my sister Victoria and my brother-in-law Patrick, your move to Florianópolis was fundamental in helping me handle this new life. Additionally, I would like to express my deepest gratitude to my girlfriend Giovanna, and her family. To my girlfriend, I want you to know that I cannot express how much joy you give me every day. My academic life would be much more stressful without you. To her family Etore, Giulia, Gladys, and Roberto thank you for all the moments we shared, which made me feel less homesick.

I would also like to thank the Bleu Studio for believing in my potential and investing in this novel project. Bleu gave me the privilege to work alongside an exceptional team composed of my close friends Fabio, João, José, and Luiza. In particular, thanks to Fabio and José for all the valuable insights on this project. In that sense, I would also like to thank my previous work experiences in Fraunhofer IPT, NEO Empreserial, and Autojun, where I could make amazing connections.

For my project advisor Alex Sandro, I want to express my gratitude for your guidance and support throughout this thesis.

Since the beginning of my studies at UFSC, I made some friends that made this experience much more enjoyable. Andre, Amir, Cassiano, Davi, Fernando K., Fernando M., Laura, Luiza Z., Matheus, Maurici, Otavio, Pedro S., Victor L., and Victor S. thank you for all the laughs that we shared.

Lastly, I would also like to thank my hometown friends Arthur, Diogo, Eduardo, Filipe, and Pedro H. All the in-person and remote moments that we spent together made me confident that I can always count on you, even if life takes each of us to a different place.

DISCLAIMER

Florianópolis, June 10th, 2024.

As representative of the Bleu Studio in which the present work was carried out, I declare this document to be exempt from any confidential or sensitive content regarding intellectual property, that may keep it from being published by the Federal University of Santa Catarina (UFSC) to the general public, including its online availability in the Institutional Repository of the University Library (BU). Furthermore, I attest knowledge of the obligation by the author, as a student of UFSC, to deposit this document in the said Institutional Repository, for being it a Final Program Dissertation (*"Trabalho de Conclusão de Curso"*), in accordance with the *Resolução Normativa n° 126/2019/CUn*.

Documento assinado digitalmente
FABIO SIMONES MENDES
Data: 31/07/2024 09:49:22-0300
CPF: ***.407.309-**
Verifique as assinaturas em https://v.ufsc.br

Fabio Simones Mendes
Bleu Studio

ABSTRACT

Since the introduction of Automated Market Makers pools as a tool to facilitate on-chain trades, various protocols have been competing to attract funds by optimizing returns for liquidity providers. Providers are crucial to the operation of liquidity pools, as they supply the tokens necessary for facilitating trades, and in return, they earn a share of the trading fees generated by the platform as an incentive for their participation. Traditionally, constant fee algorithms controlled by a governance agent are used, but these static fee structures often lead to inefficiencies, as they do not adapt fast to new market conditions. This can result in either overcharging during low-activity periods or undercharging during high-activity periods, affecting the fee collected. In addition, the pool funds are exposed to a specific type of loss known as loss-versus-rebalancing. This occurs because arbitrageurs, who aim to profit from price discrepancies, are incentivized to update the Automatic Market Maker price, extracting value from the liquidity pools. In that context, this thesis presents a study on the implementation of dynamic fees at the smart contract's level, utilizing internal pool variables to control the fee structure. The research begins by designing a simulation environment that encapsulates the primary components of the Automatic Market Maker environment, ensuring its accuracy and validity through comparison with historical data. Following this validation, three distinct dynamic fee algorithms are proposed. Each model is evaluated using a constant fee structure as a baseline. Lastly, the study advances to the implementation stage, where one of the dynamic fee algorithms is developed in a smart contract.

Keywords: Dynamic Swap Fee. Automatic Market Maker. Decentralized Finance. Smart Contracts.

RESUMO

Desde a introdução de Agentes de Mercado Automátizados como uma ferramenta para facilitar trocas de tokens na blockchain, vários protocolos têm competido para atrair fundos otimizando os retornos para provedores de liquidez. Os provedores são cruciais para a operação de fundos de liquidez, pois fornecem os montantes necessários para facilitar as negociações e, em troca, ganham uma parte das taxas de negociação geradas pela plataforma como um incentivo para sua participação. Tradicionalmente, são usados algoritmos de taxas constantes controlados por um agente de governança, mas essas estruturas de taxas estáticas geralmente levam a ineficiências, pois não se adaptam rapidamente às novas condições de mercado. Isso pode resultar em cobrança excessiva durante períodos de baixa atividade ou cobrança durante períodos de alta atividade, afetando o montante de taxas coletados. Além disso, os fundos estão expostos a um tipo específico de perda conhecido como perda contra rebalanceamento. Isso ocorre porque os arbitradores, que visam lucrar com as discrepâncias de preço, são incentivados a atualizar o preço do Agentes de Mercado Automátizados, extraindo valor dos fundos de liquidez. Nesse contexto, esta tese apresenta um estudo sobre a implementação de taxas dinâmicas implementadas no nível do contrato inteligente, utilizando variáveis internas do fundo de liquidez para controlar a estrutura da taxa. A pesquisa começa com o projeto de um ambiente de simulação que encapsula os principais componentes do ambiente de operação do Agente Automático de Mercado, garantindo sua precisão e validade por meio da comparação com dados históricos.Após essa validação, são propostos três algoritmos distintos de taxas dinâmicas. Cada modelo é avaliado usando uma estrutura de taxa constante como referência. Por fim, o estudo avança para o estágio de implementação, em que um dos algoritmos de taxa dinâmica é desenvolvido em um contrato inteligente.

Palavras-chave: Taxa de Câmbio Dinâmica. Agente de Mercado Automático. Finanças descentralizadas.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AMM	Automatic Market Maker
API	Application Programming Interface
BPT	Balancer Pool Token
CEX	Centralized Exchange
DAO	Decentralized Autonomous Organization
dApps	Decentralized Applications
DeFi	Decentralized Finance
DEX	Decentralized Exchange
ERC	Ethereum Request for Comment
EVM	Ethereum Virtual Machine
LP	Liquidity provider
LVR	Loss-versus-rebalancing
P2P	Peer-to-Peer
SQL	Structured Query Language
UFSC	Federal University of Santa Catarina
USD	United States dollar
USDC	USD Coin
WETH	Wrapped Ether

LIST OF SYMBOLS

x	Balance of first token pair
y	Balance of second token pair
k	Automatic Market Maker invariant
T	Token number
B	Token balance
W	Token weight
F	Swap fee percentage
A	Token amount
P	Pool spot price
t	Simulation step
V	Volume of swaps
L	Pool Liquidity
α_1	Decay per block of incremental volume algorithm
α_2	Gain per trade size of incremental volume algorithm
K	Reference gain
β_1	Gain of price volatility algoithm
φ_1	Gain of volume window algorithm
N	Number of blocks

CONTENTS

1 INTRODUCTION

In the context of decentralized finance, various protocols compete to attract liquidity by optimizing returns for liquidity providers (LPs). As LPs are crucial for the functioning of Automatic Market Maker (AMM), it is essential to maximize their profitability to maintain on-chain liquidity.

AMMs traditionally employ constant fee algorithms, which may lead to inefficiencies in optimizing LP returns. These static fee structures do not account for varying market conditions and can result in suboptimal fee collection, either by charging too much during low-activity periods or too little in high-activity periods. To address this issue, a dynamic contract-level fee algorithm could be implemented (LABS, C., 2023). The idea of this algorithm is to adjust fees based on real-time market conditions, potentially maximizing Liquidity provider (LP) returns by optimizing fee levels to match market demand.

1.1 MOTIVATION AND CONTEXT

Decentralized Finance (DeFi) constitutes a paradigm shift in financial services by moving away from centralized institutions like banks and brokers. DeFi eliminates intermediaries, enabling transactions directly between individuals using verifiable code on blockchain networks such as Ethereum. (SCHUEFFEL, 2021)

Within the DeFi ecosystem, decentralized exchanges created an autonomous trading mechanism with liquidity pools using AMMs to automate pricing and trading through algorithms defined in Smart Contracts. Automated Market Makers in DeFi use mathematical formulas to set asset prices based on liquidity pool ratios, enabling decentralized trading without traditional order books (XU et al., 2023). These pools reward LPs through a trading fee mechanism to encourage them to land their tokens to facilitate trades.

Although LPs earn revenue based on trading fees, the nature of AMMs exposes them to a specific kind of loss which is known as Loss-versus-rebalancing (LVR). This happens because arbitrageurs monitor pool prices and compare them to other local market prices (e.g. a Centralized Exchange (CEX) like Binance or other Decentralized Exchange (DEX)) to identify arbitrage opportunities which they could make profits from. However, in the case of DEXs, the profit earned by the arbitrator is extracted from the liquidity stocked in the pool, representing a cost to all its providers. (MILIONIS et al., 2024)

Liquidity pools commonly employ a fixed transaction fee model managed by a designated agent. This agent analyses the market conditions to operate this fee on the liquidity provider's interest. This could lead to inefficiency since the governance action frequency is significantly lower than the trading frequency.

1.2 BLEU STUDIO

This thesis is part of Bleu's internal initiative to optimize liquidity utilization in the blockchain ecosystem. Bleu is a collaborator in web3 technology and user experience, bringing together the needs of both enterprises and Decentralized Autonomous Organizations. The organization is driven by the belief in the immense potential to enhance and simplify blockchain interactions. Bleu's mission is to bridge the gap between the extensive capabilities of blockchain technology and the current user experience in web3, which is widely recognized as having significant room for improvement (GATEWAY.FM, 2023). The origins of Bleu are rooted in the entrepreneurial spirit of alumni from the Control and Automation Engineering program at the Federal University of Santa Catarina (UFSC).

Founded on December 2022, Bleu was established to make substantial contributions to software projects within the cryptocurrency environment. The founding members, already well-acquainted with one another, shared a unified vision: they believed their combined knowledge could drive significant progress within the ecosystem by delivering exceptional solutions. Despite being in its nascent stages and being composed of a small team of six members, Bleu has already showcased its expertise across various sectors, including web3, web2, and web2.5. (DAO, 2023)

1.3 OBJECTIVE OF THE THESIS

1.3.1 General objective

The general objective of this thesis is to develop a dynamic fee algorithm that optimally adjusts transaction fees in AMMs to improve LP incentives and market stability. Managing transaction fees in AMMs is a complex task because setting lower fees can attract higher trading volumes, thus increasing the overall fee collected. However, lower fees might not be enough to compensate LPs for their risks. On the other hand, higher fees can generate more revenue per trade but may deter traders, reducing overall volume.

The central problem addressed by this work is the challenge of controlling transaction fees to both protect LPs from significant losses due to arbitrage and maximize their returns. Arbitrage, while necessary for price alignment, can extract a large amount of funds from LPs. The dynamic fee model needs to effectively balance these competing interests by increasing the fee during periods of high trading activity or volatility, thereby reducing the impact of arbitrage, and lowering fees when trading activity is low to encourage more trades.

In summary, the goal is to protect LPs from excessive fund losses due to arbitrage while maximizing their returns, making liquidity provision more attractive and sustainable. By implementing an adaptive fee structure, this work aims to enhance the efficiency and stability of AMMs, drawing more participants to provide liquidity on-chain.

1.3.2 Specific objectives

The project aims to show conditions where a dynamic fee approach will lead to optimal liquidity providers' returns. To achieve this, the specific objectives of this work can be described as:

- Design a computational model that can simulate an AMM to test new swap fee models;
- Analyze different automated fee systems and find at least one that can outperform the constant fee mechanism;
- Create a proof of concept smart contract implementation of the dynamic fee approach leveraging an existing AMM.

1.4 DOCUMENT OUTLINE

The document begins with the Theoretical Background chapter, providing the technical information necessary to understand the problem and the proposed solution. This section includes an overview of AMMs, liquidity provision, fee structures, and the behavior of the main agents of the DeFi environment.

Following this, the Model Development chapter details the development and testing of the simulation used to evaluate new fee models. It explains the design of the testing environment, the parameters and assumptions chosen, and the methods used to validate the simulation results against real-world data.

The next chapter introduces and evaluates three proposed fee models designed to optimize LP returns. Each model's algorithm is explained in detail, followed by an analysis of its performance using the developed simulation framework. Comparative results with a constant fee model are also presented.

In the Smart Contract Development chapter, the design and implementation of smart contracts using the Balancer V2 protocol are covered. This section explains how the proposed fee models are integrated into the smart contracts, leveraging Balancer V2's interface to enable new mechanisms in custom pools.

Lastly, the conclusion summarizes the work, presenting the key findings. It also discusses potential future work, suggesting areas for further research to enhance the efficiency and effectiveness of AMMs in decentralized finance.

2 THEORETICAL

This chapter reviews the theoretical concepts that are fundamental to the development and understanding of the thesis. The first section reviews blockchain focusing on the Ethereum context. In the next one, an overview of the decentralized finance system is presented. In the third section, the text dives into the details of AMMs. Lastly, similar studies of dynamic fees in the context of AMM are presented.

2.1 BLOCKCHAIN

Blockchain technology logs transactions in blocks, much like a linked digital ledger of record books. A new block starts when the previous one is filled, and they are all connected sequentially to create a chain. The blockchain's integrity and sequential correctness are guaranteed by the distinct cryptographic references included in each block to their predecessors. This idea was originally presented for secure document timestamping (HABER STUART STORNETTA, 1991). The Bitcoin whitepaper, written by the pseudonymous Satoshi Nakamoto popularized it by proposing a decentralized finance system (NAKAMOTO, 2008).

The system is distributed and managed by a Peer-to-Peer (P2P) network in which every participant in the network retains a copy of the ledger. Modifying any previously recorded information within a blockchain is exceedingly difficult, as it would require altering all subsequent blocks in the chain, such a task is computationally impractical under normal circumstances.

In essence, blockchain is a groundbreaking technology for record-keeping that offers enhanced security, and transparency. In addition, this technology can operate independently of any central authority.

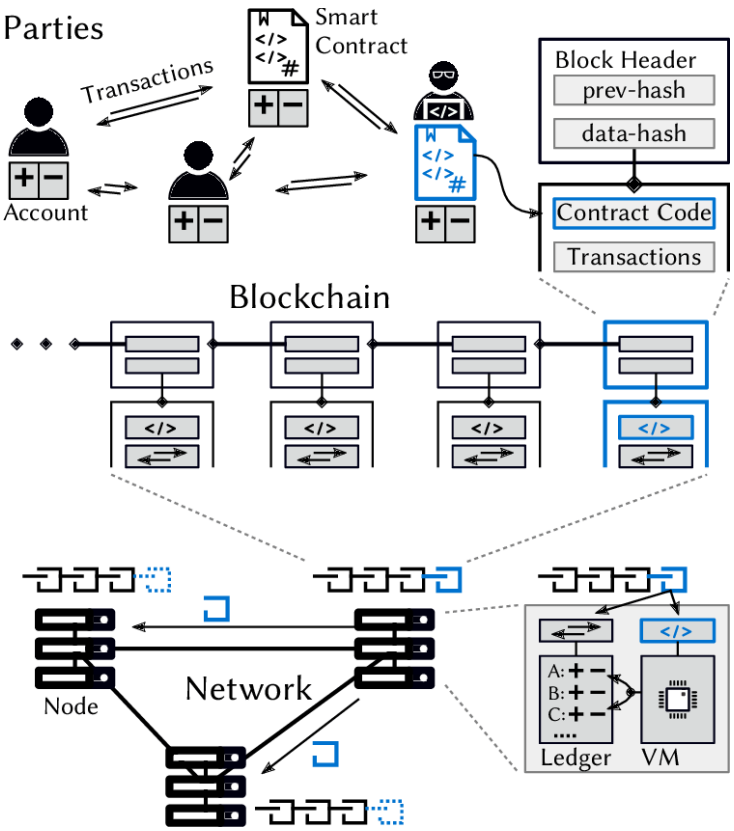
2.1.1 Ethereum

Further innovations were introduced with the development of Ethereum, extending the capabilities of blockchain with the integration of smart contracts (BUTERIN, 2014). This innovation has expanded the utility of blockchain beyond just one currency transaction, enabling a more complete financial system. These contracts have the ability to control and save the states of the blockchain, as illustrated in Figure 1. The figure shows different parties or users interact with each other and smart contracts by doing transactions, that are computed by the network nodes.

Ethereum smart contracts are self-executing computer programs where the terms of the agreement are directly written into lines of code. Once deployed, they become immutable and autonomous, meaning they cannot be altered and will execute automatically. This immutability ensures transparency and security, as all parties involved can trust that the contract will function as intended without interference.

The execution of Ethereum smart contracts is facilitated by the Ethereum Virtual

Figure 1 – Ethereum arcithecture



Source: (REGNATH; STEINHORST, 2018)

Machine (EVM), which allows these contracts to operate in a decentralized environment. This decentralization is crucial as it ensures that the contract's execution is verifiable and resistant to censorship or downtime. Smart contracts can handle a wide range of functions, from simple transactions to complex Decentralized Applications (dApps) involving multiple parties. They have been extremely important in the progress of DeFi, enabling activities such as lending, borrowing, and trading without traditional financial institutions. (ZHOU H MILANI FARD A, 2022)

Smart contracts on Ethereum are usually written in Solidity, a programming language designed specifically for this reason. The language supports various features such as inheritance, libraries, and complex user-defined types, allowing the creation of sophisticated and secure contracts. (SIMPLELEARN, 2023)

Despite their innovative capabilities, smart contracts have significant security risks. These risks stem from various vulnerabilities in the smart contract code (GRISHCHENKO; MAFFEI; SCHNEIDEWIND, 2018). This rigidity has led to significant financial losses, as seen in notorious incidents like the Decentralized Autonomous Organization (DAO) attack and the Parity multi-sig wallet breaches (SIEGEL, 2016). To mitigate these risks, rigorous security audits and the use of formal verification methods are essential to identify and address potential vulnerabilities before deployment.

2.1.2 Smart contract gas consumption

Ethereum charges a fee called "gas" for processing transactions and running smart contracts on its blockchain. Gas optimization is crucial in smart contract development due to the direct impact it has on the cost of executing transactions on the blockchain. Every operation within a smart contract consumes a certain amount of gas, which users must pay for in the form of cryptocurrency. High gas costs can deter users from interacting with a contract, limiting its adoption and utility. (TEAM, I., 2024)

2.2 DECENTRALIZED FINANCE OVERVIEW

Built on top of blockchains, DeFi represents a shift in the financial sector, moving away from traditional centralized institutions like banks and brokers. Using blockchain technology, DeFi allows financial processes to be conducted directly between parties through smart contracts. With that, DeFi aims to improve the accessibility, transparency, security, and autonomy of the financial system. (SCHUEFFEL, 2021)

2.2.1 Cryptocurrencies

Cryptocurrencies, at their core, represent various forms of digital assets and tokens operating on blockchains. These digital assets can vary significantly in their purposes, structures, and functions within the crypto ecosystem. For the objective of this work, it is possible to divide the cryptocurrencies into three types:

- **Native coins**: The more basic type of cryptocurrency, which is built on their independent blockchains, to pay for the computational work of validating the blocks of the chain. ETH is the native token of the Ethereum blockchain; (KRYPTOMAT, 2024)
- **ERC-20 tokens**: Another type of token that exists on the Ethereum blockchain is the ERC-20 tokens which are a standardized type of smart contract that represents a token. An Ethereum Request for Comment (ERC) is a technical standard used for smart contracts on the Ethereum blockchain. This standard defines a common list of rules that an Ethereum token has to implement, ensuring compatibility with the broader Ethereum ecosystem. Unlike native tokens, ERC-20 tokens benefit from the security and functionality of the Ethereum blockchain without needing to establish their own network; (VOGELSTELLER; BUTERIN, 2015)
- **Stablecoins**: In addition, stablecoins are a type of cryptocurrency designed to minimize price volatility by being pegged to a more stable asset, typically a fiat currency like the United States dollar (USD). (COINBASE, 2024)

2.2.2 Exchanges

A cryptocurrency exchange is a platform that facilitates the buying, selling, and trading of cryptocurrencies. These exchanges serve as a crucial intermediary in the crypto ecosystem, providing a marketplace where users can trade cryptocurrencies for other digital currencies or traditional fiat money. The operations of these platforms are similar to traditional stock exchanges, where buyers and sellers meet to conduct transactions based on current market prices.

Exchanges play a pivotal role in the liquidity and price discovery of cryptocurrencies. They aggregate trade orders from participants, establishing market prices through supply and demand dynamics. This operation enables individuals and institutions to speculate on price movements, invest in digital assets, and manage financial risk through diversification. By offering a structured environment for transactions, exchanges also contribute to the broader adoption and accessibility of cryptocurrencies, making them integral to the digital asset economy. (CENTIC, 2023)

It is possible to divide exchanges in the cryptocurrency ecosystem into two types:

- **Centralized Exchanges**: It operates under the control of a central authority that facilitates and oversees all transactions. Users of CEXs transfer their crypto assets to the exchange, which then provides internal mechanisms for orders to be executed. The benefits of using a CEX include a higher trading speed and higher liquidity;
- **Decentralized Exchanges**: It operates without a central authority and enables direct P2P transactions on a blockchain. To do that, DEXs rely on smart contracts to automate and secure transactions. The main advantage of DEXs is user privacy and that the user maintains full custody of its assets.

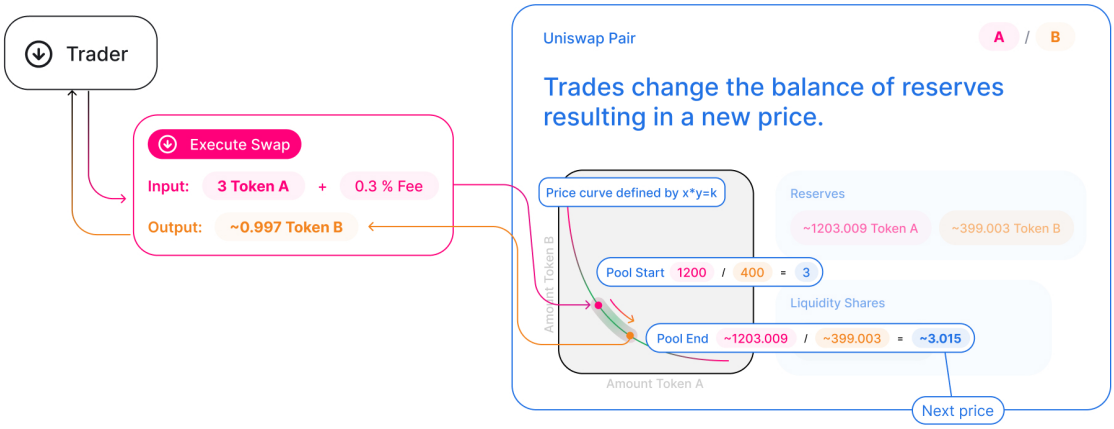
2.3 AUTOMATIC MARKET MAKERS

Automated Market Makers are a foundational component in DeFi, providing a mechanism for trading digital assets without needing traditional order books. Unlike conventional exchanges where buyers and sellers place orders to determine prices, AMMs utilize mathematical formulas to set the price of assets based on the ratio of tokens in a liquidity pool, a contract that holds the tokens and implements the AMM logic to facilitate the trades. This method ensures continuous liquidity, as trades can always be executed. Popular AMM protocols like Uniswap, Curve, and Balancer use different formulas, such as the constant product formula described in equation (1), where X and Y represent the quantities of two different assets in the pool, and K is a constant value that remains unchanged after trades. (XU et al., 2023)

$$k = X * Y \tag{1}$$

This invariant ensures that any trade impacting the pool will adjust the quantities of X and Y such that their product remains constant. When a trader wishes to swap one asset for another, the pool automatically recalculates the new quantities of each asset to maintain the constant product as presented in Figure 2. This mechanism inherently makes the pool's asset ratios follow the external market price because of arbitrage opportunities as will be described in the subsection 2.3.3.

Figure 2 – Trade on a constant product AMM



Source: (UNISWAP, 2020)

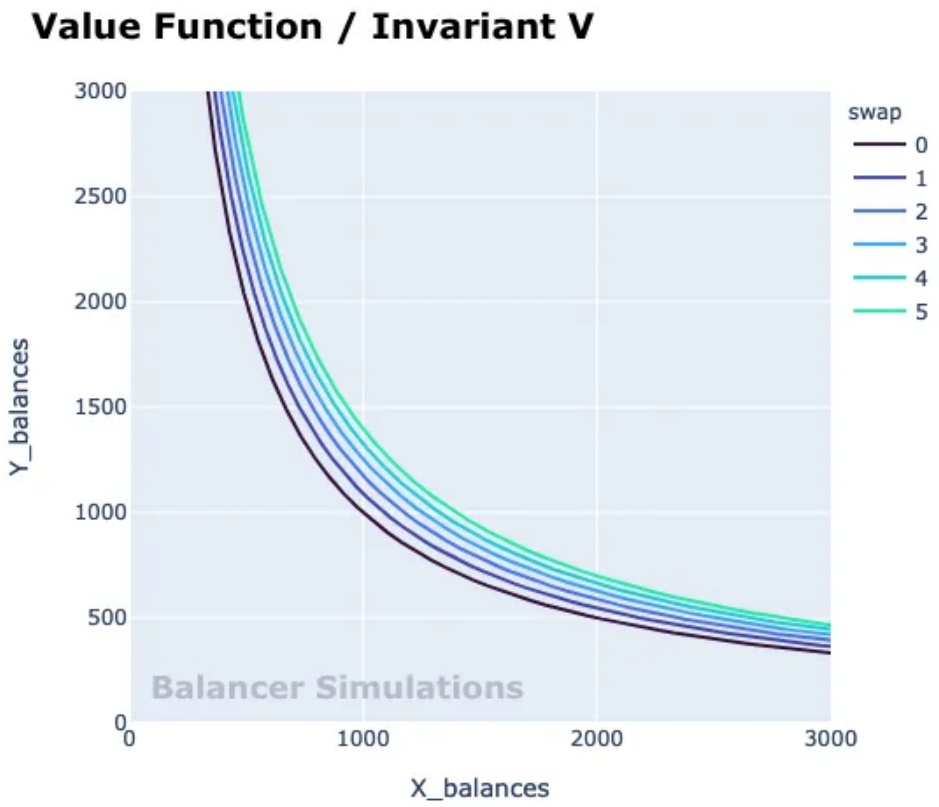
The swap fee is applied to the input token, meaning the fee is subtracted before the swap happens. The fee structure is intimately linked to the invariant curve, which governs the price relationship between the tokens in the pool. In that sense, swaps inflate the invariant curve, meaning an inflation on the pool share price as well. This behavior is represented in Figure 3.

2.3.1 Liquidity Providers (LPs)

LPs supply the necessary tokens by depositing them into liquidity pool contracts, enabling traders to execute transactions without relying on a traditional order book. LPs are encouraged through the distribution of trading fees generated by the pool. Every time a trade occurs, a fee is charged, which is then proportionally allocated to the LPs based on their share of the total pool. This provides a constant income stream, making it an attractive proposition for investors looking to earn passive income.

One of the significant advantages of being an LP in an AMM is the passive balancing of the portfolio. By providing liquidity to a pool, LPs essentially automate the process of maintaining a balanced portfolio between the two assets in the pool. As trades occur, the pool's composition changes and the LPs' share adjusts accordingly. This means LPs do not need to manually rebalance their holdings, as the AMM protocol will handle this process by following the price market due arbitrages. (XU et al., 2023)

Figure 3 – Invariant curve inflation over swaps



Source: (KREITENWEIS, 2021)

AMMs can have different fee structures to manage the charged value and how it is received by the LP. Some AMMs have immutable trading fees, which are set at a fixed percentage and cannot be altered once the pool is deployed. This provides predictability for liquidity providers and traders (ADAMS; ZINSMEISTER; ZINSMEISTER, 2024). Other AMMs enable trading fee changing, which can be adjusted by a governing entity or through decentralized governance mechanisms (MARTINELLI, 2021b). This flexibility allows the protocol to adapt to changing market conditions, optimize liquidity incentives, and respond to community feedback. However, mutable fees also introduce the risk of centralization and potential manipulation.

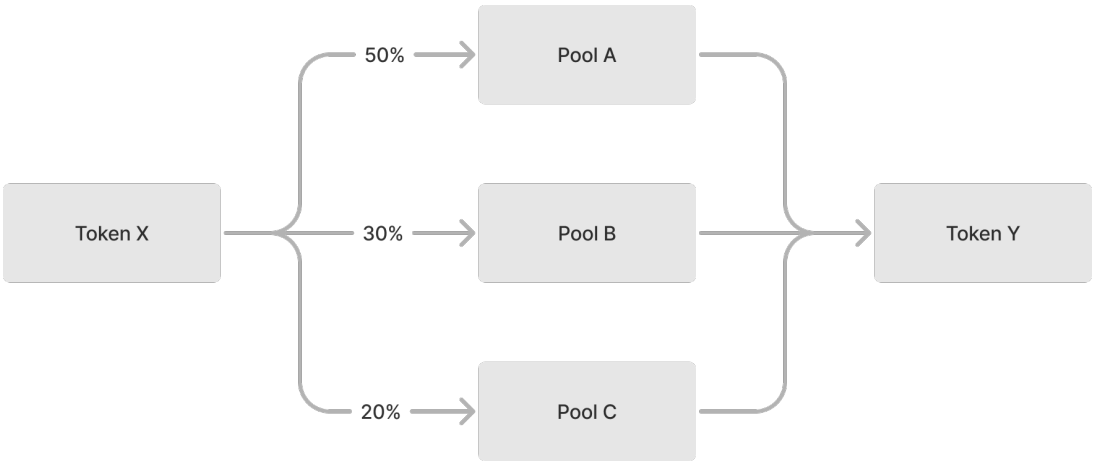
2.3.2 Aggregators

In the DeFi ecosystem, DEXs host multiple liquidity pools for the same token pairs. Each pool can have varying levels of liquidity, pricing, and transaction fees, making it challenging for users to find the best trading route. This fragmentation can lead to inefficiencies and suboptimal trade executions for users who manually search for the best prices across different DEXs.

DEX aggregators address this issue by integrating with multiple DEXs and their respective liquidity pools. These aggregators utilize algorithms to scan the available liquidity across various platforms, ensuring that users receive the best possible on-chain price for their

trades. By aggregating liquidity from multiple sources, these platforms also has the ability to split orders across different pools to minimize slippage and maximize the efficiency of each transaction as shown in Figure 4. This integration not only simplifies the trading process for users but also enhances liquidity and price discovery within the DeFi ecosystem, providing a more seamless and cost-effective trading experience. (COINMARKETCAP, 2024)

Figure 4 – Example of trade aggregator route



Source: Author

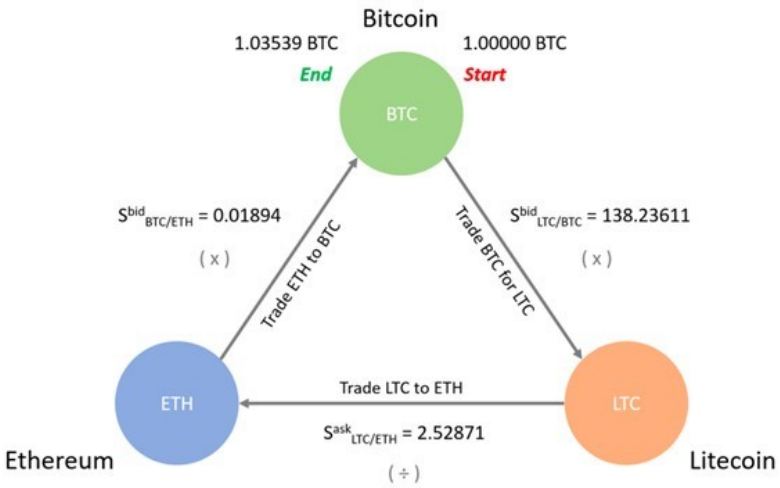
2.3.3 Arbitrage

Arbitrage involves exploiting price discrepancies between different trading platforms to secure risk-free profits. In the context of AMMs, arbitrageurs capitalize on price differences between DEXs and CEXs or among multiple DEXs. When the price of a token pair on an AMM deviates from its market price on a CEX or another DEX, arbitrageurs can buy the underpriced asset and sell it where it is overvalued as presented in Figure 5. This activity helps align prices across markets, ensuring that the token prices in AMMs reflect the market conditions. In the context of this work, it is possible to divide arbitrages into two types: (RODRIGUEZ, 2023)

- **On market price move**: This opportunity happens when the token prices themselves change, due to any market expectation or condition. This kind of opportunity is influenced by the operational dynamics of blockchain networks. On AMMs, transactions are processed in the network blocks, in Ethereum each new block is created around every 15 seconds. The block frequency is relatively slow compared to the millisecond-level trade execution in CEXs. This represents that arbitrages can take advantage of this characteristic by updating outdated AMM prices;
- **On pool price move**: If a large trade happens the pool price can move away from the market reference, which creates an arbitrage opportunity. In addition, there is a kind of crypto attack that uses a similar concept of this arbitrage, when an uninformed trade selects unsafe parameters to trade on the AMM. This attack is called a sandwich

attack and it is a type of market manipulation where a malicious trader places two transactions around a victim's transaction - one before to drive up the price, and one after to sell at the higher price, profiting from the price difference at the victim's expense. (SERGEENKOV, 2021)

Figure 5 – Example of triangle arbitrage



Source: (RODRIGUEZ, 2023)

All-in-all, arbitrage plays a critical role in maintaining the efficiency and liquidity of AMMs. The continuous arbitrage activity ensures that prices remain competitive and reflective of the actual market value, benefiting on-chain trading activity. (RODRIGUEZ, 2023)

However, if the arbitrageur is profiting, it is plausible to argue that this value is being extracted from liquidity providers. Even after an uninformed trade, the funds remain in the AMM for a moment. This process, known as loss-versus-rebalancing (LVR), represents the cost that liquidity providers incur during each rebalance of their funds by arbitrageurs. As presented on (FRITSCH; CANIDIO, 2024), it is estimated that this cost often overcomes the fees earned by LPs removing their incentives to provide liquidity to AMMs.

2.4 BALANCER V2

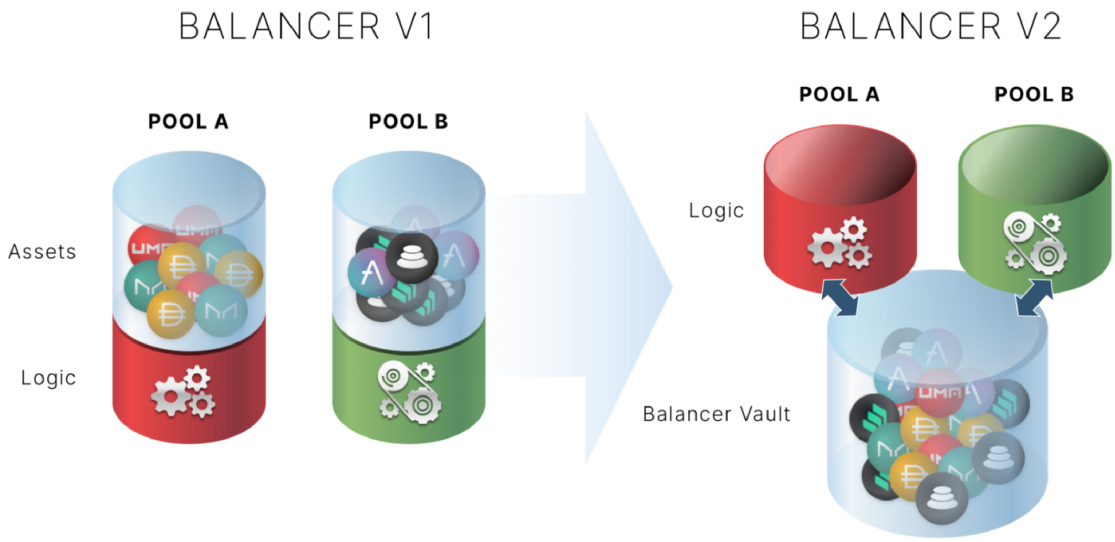
Due to the Balancer and Bleu partnership, this thesis will use the Balancer context for all the development (DAO, 2023). Beyond a DEX, Balancer Protocol is an open-source, decentralized platform built on the blockchain that functions as an automated portfolio manager and liquidity provider. It allows users to trade ERC-20 tokens, invest in liquidity pools, and earn yields from trades in a trustless and permissionless way. With its unique approach to automated market making and liquidity provision, Balancer is designed to enhance user accessibility and optimize the crypto trading experience. (LABS, B., 2021)

Balancer V1 was the initial release of the Balancer Protocol that introduced the concept of liquidity pools that could hold multiple tokens with an AMM that could handle varying

weightings. This flexibility allowed users to fully control how exposed they would be to each token pool. Balancer V1 enabled liquidity providers to benefit from continuous rebalancing, where the protocol's unique mechanics ensured that traders, by conducting swaps, effectively maintained the desired pool ratios. (LABS, B., 2024)

Balancer V2 introduces enhancements to the Balancer Protocol, aimed at improving security, flexibility, capital efficiency, and gas efficiency. As is possible to visualize in Figure 6, V2 changed to a single Vault architecture, which consolidates the management of all assets deposited into Balancer Pools in a single contract. This architecture separates token accounting and management from the pool logic, enabling simplified pool contracts and facilitating pools to implement any customized AMM logic. By reducing the complexity of individual pool contracts, the Vault architecture enhances the overall efficiency and security of the protocol.

Figure 6 – Balancer V2 Vault



Source: (LABS, B., 2024)

In that sense, Balancer V2 introduces permissionless and customizable AMM logic, allowing developers to create pools with tailored functionalities. This flexibility supports a wide range of pool types, including Weighted, Stable, Liquidity Bootstrapping, and Protocol Pools, each catering to different use cases and asset configurations.

The interface for performing swaps, as well as joining and exiting pools, is through the centralized Vault architecture. The Vault acts as the single point of entry for all asset management activities. When a user initiates a swap, the Vault receives the input tokens and processes the transaction according to the pool's specific AMM logic. The Vault then adjusts the token balances within the respective pools and distributes the output tokens to the user. This centralized processing significantly reduces gas costs and complexity compared to previous decentralized approaches, enhancing the user experience by providing a more straightforward and cost-effective method for executing trades.

2.5 SIMILAR DYNAMIC FEE MODELS STUDIES

Since the start of DeFi, numerous researchers have focused on studying the losses incurred by LPs in AMMs. Initially, impermanent loss was identified as the primary form of these losses, characterized by a reduction in the value of staked assets compared to simply holding them. However, in 2022, the concept of impermanent loss was refined and replaced with the notion of LVR (MILIONIS et al., 2024). While impermanent loss uses a holding strategy as a reference, the concept of AMMs involves continuous portfolio rebalancing. In that sense, LVR, which quantifies the losses LPs incur from rebalancing their portfolios over time, provides a more accurate representation of the financial impact of the arbitrage action on LP funds.

In that sense, dynamic fee emerges as one of the possible solutions for LVR. The DEX Balancer, in 2021, has teamed up with an external team to maximize returns on the second version of their protocol dynamic changing the swap fee based on off-chain simulation. The idea was to use battle-tested techniques to identify and update the swap fee on each period of time. However, the initiative is not running anymore on the Balancer protocol. (MARTINELLI, 2021a)

Cata Labs (LABS, C., 2023) conducted a comprehensive study on the implementation and potential returns of dynamic fees within the Uniswap V4 protocol. Given that Uniswap maintains multiple pools with different swap fee percentages but identical trading pairs, the initial phase of the study focused on identifying which swap fee maximized rewards within different time windows. Subsequently, the theoretical returns of a dynamic fee pool, capable of consistently applying the optimal fee for each time window, were compared with the actual fees collected from each individual pool, showing that a dynamic fee model can outperform the fixed one. The study further explored various dynamic swap fee models, and Figure 7 encapsulates the benefits and trade-offs associated with each model.

Additionally, Atis E (E, 2023), in collaboration with Brokkr Finance, has conducted research into the implementation and potential returns of dynamic fees within the Uniswap V4 framework. The primary objective of this research was to understand the relationship between the liquidity of asset pools, the volatility of asset pairs, and the corresponding fees collected. To achieve this, a simulation model was developed to emulate the interactions between a CEX and an AMM, specifically focusing on the behavior of arbitrageurs engaging in pool swaps. Next, the model incorporated the effects of uninformed flow, representing random swaps within the pool. The author concludes that the optimal fee is highly influenced by two key factors: asset volatility and the volume of uninformed flow.

Figure 7 – Summary of advantages and tradeoffs of different fee models

Design	Advantages	Tradeoffs
Multivariate Model for Fees	1. Captures fee-generating signal across a wide range of factors 2. Outperforms the strategy of selecting the best Uniswap fee tier	1. Requires additional computational power and would need infrastructure like coprocessors to implement 2. Doesn't discriminate between informed and uninformed flow
Volatility dependent Fees	1. Can drive additional volume to pool in periods of low volatility 2. Compensates LPs better/drives away volume in periods of high volatility	1. Can become prohibitively expensive for swappers 2. Doesn't discriminate between informed and uninformed flow
Constant Volatility based Fees	1. The AMM incentivises frequent and large trades. 2. Creates a ripple effect of arbitrage trades	1. LPs collect very little fees for high-value swaps 2. DEX aggregation tools might not recommend this route compared to similarly sized LP pools
Pool imbalance-based fees (Curve V2)	1. Simple design, one of the only mechanisms already in use 2. Fees are adjusted to increase LP profitability and ultimately improve the swap quotes	1. Complexity of pool setup is very high 2. Doesn't discriminate between informed and uninformed flow
Flow driven fees	1. Simple design, easy to implement 2. Disincentives repetitive trades in same direction (captures signal from trades in previous block)	1. Improvement over existing fee models is limited (~8-10%)
Flow identification	1. Fees for uninformed traders can be minimised to offer better execution 2. LPs get to capture fees mainly from uniformed traders and thus face lower LVR/markout losses	1. Known methods exist to avoid detection as toxic flow via new wallet creation, trade splitting 2. Another computationally heavy mechanism & might require off-chain compute 3. Price discovery might be slower

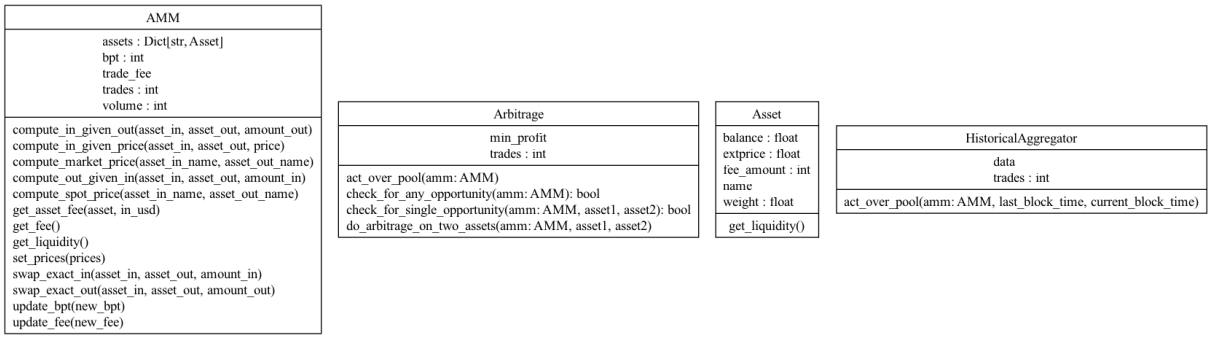
Source: (LABS, C., 2023)

3 MODEL DEVELOPMENT

The objective of this chapter is to develop a computational model that can simulate the AMM behavior for the evaluation and testing of fee algorithms. Initially, the scenario of the simulation will be described followed by an explanation of all data used and how it was acquired. Next, the pool initialization method will be described. After that, a model simulating AMM behavior will be elaborated. In addition, a model simulating arbitrage opportunities and their interaction with the AMM will be described, followed by the construction of a model representing the functionality of the aggregators. Lastly, the entire model framework will be validated against historical data to ensure its reliability and accuracy in representing real-world scenarios.

All components of the model will be implemented using Python language using object-oriented models to ensure modularity and clarity. The class diagram is illustrated in the Figure 8.

Figure 8 – Model components class diagrams



Source: Author

At each step of the simulation, the arbitrageur and the aggregator can act on the pool. The priority is of the arbitrageur, which means that it acts first on each simulated block to take advantage of arbitrage opportunities.

- **Asset class**: represent the individual tokens involved in the trading pair, encapsulating properties such as current market price, and balance;
- **AMM class**: represent the liquidity pool itself, including functionalities for executing swaps, and adding or removing liquidity;
- **Arbigrate class**: represent the arbitrageurs that will try to extract value on the pool by updating the pool price to the market reference;
- **Historical Aggregator class**: represent the aggregator trades and if it will or will not be routed through the pool.

3.1 SIMULATION CONSTANTS

The simulation will be built on top of the Balancer DEX. This specific protocol has been chosen due to a strategic partnership between Bleu and Balancer for project development. In addition, since the second version launch, Balancer has incorporated custom pool models, that can be used to implement the resulting automatic fee algorithm proposed in this thesis. (KOZLOVSKI, 2021)

The ETH-USD pair was selected due to its prominence as one of the largest trading pairs on the blockchain, ensuring substantial trade volume and liquidity. Then, the B-50USDC-50WETH pool on Balancer was defined as the reference, since it is one of the biggest pools on the DEX for this pair. (BALANCER, 2021)

This pool implements the weighted constant product invariant of Balancer, a variation of the constant product invariant defined as equation (2) to support multiples tokens with different weights, where T represent the number of tokens, B represent the amount of each token locked on the pool, and W the pool weight. This weight was created to shift the portfolio rate of the pool. However, since the pool just has two tokens and the weights are both equal to 50, the formula is the same as the regular constant product defined in equation (1).

$$k = \prod_{t} B_T^{W_T} \tag{2}$$

One of the pool tokens is the Wrapped Ether (WETH) a token that represents ETH on the ERC-20 standard, making it compatible with various decentralized applications (BINANCE, 2022). The other one is the USD Coin (USDC), a stablecoin pegged to the US dollar, providing a stable value reference point within the volatile crypto market. (CENTRE, 2018)

In addition, multiple pools of Balancer, including the one that will be used for this simulation, started to enable discounted swap fees for aggregator agents. This program started in May of 2023 (SOLARCURVE, 2023). For simplicity, we will not use data above this date. To decrease data size storage, we will use 1 year of data.

3.2 DATA ACQUISITION AND TRANSFORMATION

To ensure the simulation of the AMM is as realistic as possible, a combination of on-chain and off-chain data will be acquired.

3.2.1 Off-chain data

The off-chain data will be utilized to provide historic price information for the tokens involved in the transactions. This data will be mainly used by the arbitrageur agent to visualize arbitrage opportunities.

3.2.1.1 External market price

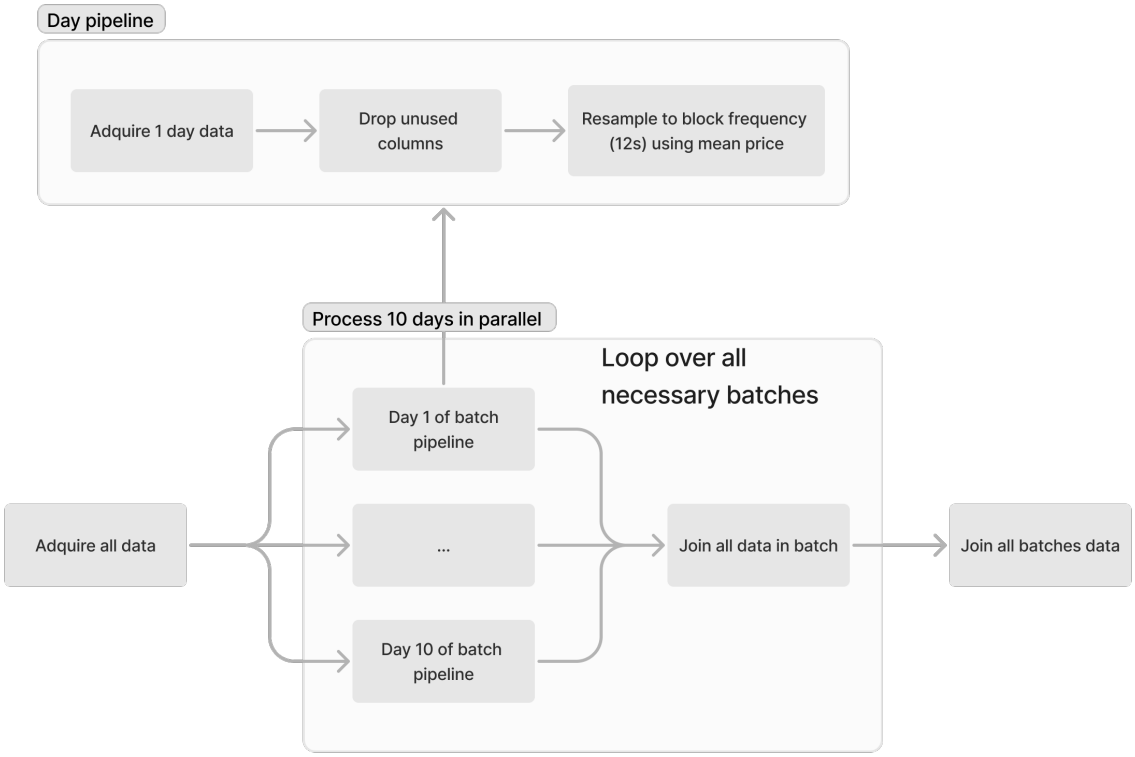
As an index of the market price, Binance API data will be used. Binance Application Programming Interface (API) allows developers to programmatically access and interact with the Binance cryptocurrency exchange. The acquired data can be represented on tabular data with simplified versions of the columns described in table 1.

Name	Description
open_time	Period start timestamp (UTC)
open	Price on the start of the period
close_time	Period close timestamp (UTC)

Table 1 – Binance API historical price simplified version of columns

The data was acquired in a one-second interval, which made most of the periods have the same value for all price columns. Therefore, for simplification reasons, it was considered just the *open_price*. The Binance API has limitations on results per call and calls per second, forcing the acquisition to be made in batches. The acquisition pipeline is illustrated in Figure 9.

Figure 9 – Historical market price data acquisition pipeline



Source: Author

3.2.2 On-chain data

On-chain data will be used for the aggregator trades, pool liquidity, and historical swap fees.

The aggregator data will be acquired using Dune Analytics. Dune Analytics is an open-source platform that enables users to query, analyze, and visualize blockchain data using Structured Query Language (SQL). (TEAM, D. A., 2024)

The historical swap fee and pool liquidity will use the Balancer subgraph. A subgraph is a specialized queryable data structure used to index and organize blockchain data, making it easily accessible and analyzable. Subgraphs extract data from blockchain events, transform it into a structured format, and store it in a format that can be efficiently queried using GraphQL. (JONES, 2023)

3.2.2.1 Pool snapshots

One of the available data of Balancer subgraph is the Pool Snapshot. This table represents daily snapshots of the pools. Each day the fields described in table 2 are stored in the database.

Name	Description
pool	Balancer pool identification
amounts	Amount of each token locked on the pool
totalShares	Balancer Pool Token (BPT) amount
swapFees	Total value of fees generated on swaps (USD)
liquidity	Total liquidity locked on the pool (USD)
timestamp	Snapshot timestamp (UTC)

Table 2 – Fields of *PoolSnapshot* table on Balancer subgraph

Since the simulation period was lower than the subgraph API limit, we could acquire the data in one call. In that sense, the data acquisition is illustrated in Figure 10. Since this data is in a daily frequency, forward fill was used to interpolate the data in a block frequency.

Figure 10 – Pool snapshot data acquisitions pipeline

Source: Author

3.2.2.2 Swap Fee Updates

Each time the pool changes its swap fee, it emits an event that is stored by the Balancer subgraph. This data stored has the fields described in the table 3.

Since the pool changed its swap fee a few times in the simulation period, the data acquisition was very similar to the pool snapshot data, illustrated in Figure 10.

Name	Description
scheduledTimestamp	Timestamp to update the swap fee
amounts	Amount of each token locked on the pool
totalShares	BPT amount
swapFees	Total value of fees generated on swaps (USD)
liquidity	Total liquidity locked on the pool (USD)
timestamp	pool snapshot timestamp (UTC)

Table 3 – Fields of *SwapFeeUpdate* table on Balancer subgraph

3.2.2.3 Aggregators trade history

The aggregator's trade history will be acquired from Dune Analytics. Dune Analytics is a platform that enables users to query blockchain data. It operates by indexing data from various blockchain networks, transforming it into a structured and queryable format using SQL. The platform connects directly to blockchain nodes and continuously updates its dataset, ensuring that the data remains current and accurate. (TEAM, D. A., 2024)

Dune Analytics provides a pre-existing table that contains comprehensive data on all decentralized exchange trades. To utilize this resource, we will acquire the data filtering to include only swaps involving USDC and WETH. This filtered data will then be downloaded and transformed locally to suit our analytical needs. The downloaded data format is represented by the table 4.

Name	Description
project	DEX or aggregator protocol name
block_time	Block timestamp of the trade (UTC)
token_bought_symbol	Symbol of the token bought
token_bought_amount	Amount of token bought
token_sold_symbol	Symbol of the token sold
token_sold_amount	Amount of token sold

Table 4 – Fields of acquired data of Aggregators

3.3 AMM

The introduction of the AMM agent will implement its core functionalities, structured into three key sections: initialization, swap, and liquidity change. The initialization section will cover the setup and configuration of the AMM agent at any moment of the simulation. The swap section will present the mechanics of executing trades within the pool. Lastly, the liquidity changing section will explain how the AMM agent manages the addition and removal of liquidity since it impacts the pool's effective price.

3.3.1 Initialization

The initialization method was developed to start the pool at any simulation point. For that, it is considered the last snapshot of BPT and amounts. This data was chosen because these variables are the ones being managed by the pool, while the liquidity is inferred by market price indicators.

3.3.2 Swap

The swaps on the AMM will follow its invariant, defined in equation (2). Three separate math functions can be derived from the pool invariant that calculates different types of swaps depending on the reference used by the trader. All math functions are defined on equations (3)-(5), where F is the swap fee percentage, A the amounts of the trade, B the pool balances before the trade, W the pool weights and P the pool spot price.

- **In given out**: The AMM calculates how many tokens you need to send to the pool to receive a specific amount back. This is described by equation (3). Can also be called as buy order.
- **Out given in**: The AMM calculates how many tokens you'll receive from the pool when you send in a specific amount. This is described by equation (4). Can also be called as sell order.
- **In given price**: This is a helper function for arbitrageurs. The AMM calculates how many tokens to send to the pool to move its price to a specific value. This is described by equation (5).

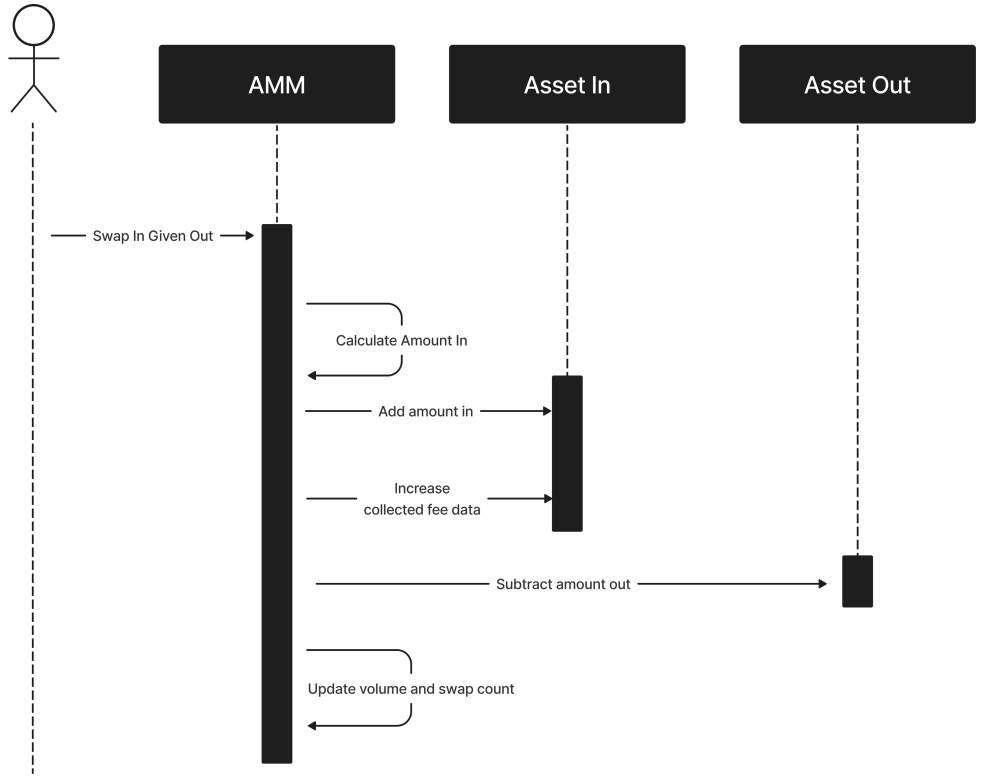
$$A_{in}(A_{out}) = \frac{B_{in}}{1-F}\left[\left(\frac{B_{out}}{B_{out}-A_{out}}\right)^{W_{out}/W_{in}} - 1\right] \tag{3}$$

$$A_{out}(A_{in}) = B_{out}\left[1 - \left(\frac{B_{in}}{B_{in} + (A_{in} * (1-F))}\right)^{W_{in}/W_{out}}\right] \tag{4}$$

$$A_{in}(P_{desired}) = B_{in}\left[\left(\frac{P_{desired}}{P_{current}}\right)^{W_{out}/(W_{out}+W_{in})} - 1\right] \tag{5}$$

With that, two swap methods were defined, with the fixed in and out amount. Each Asset class will also save the fee collected, which for simplification reasons will be transformed into dollars with the current market price. A sequence flow of a buy order is illustrated in Figure 11.

Figure 11 – Buy order sequence diagram

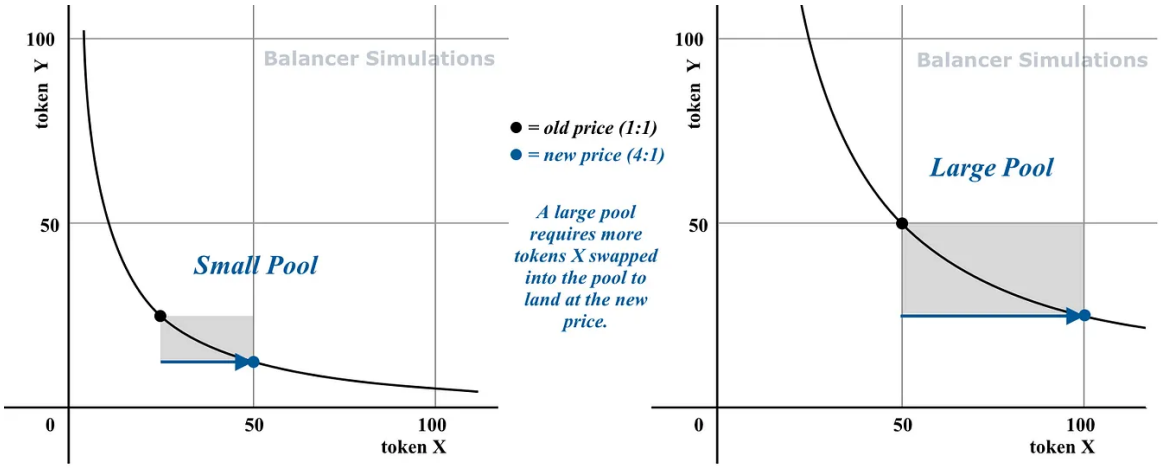


Source: Author

3.3.3 Liquidity addition and removal

As shown in Figure 12, the effective price of a pool in an AMM depends significantly on its liquidity. High liquidity in a pool means that it can handle large trades with minimal price impact, resulting in more stable and favorable trading conditions. This relationship can be visualized on the math function (6).

Figure 12 – Impact of liquidity on amount to move pool price



Source: (KREITENWEIS, 2021)

In the model, we will utilize pool snapshots to update the liquidity of the pool by monitoring the BPTs. As this value serves exclusively to enhance the realism of the swaps, we

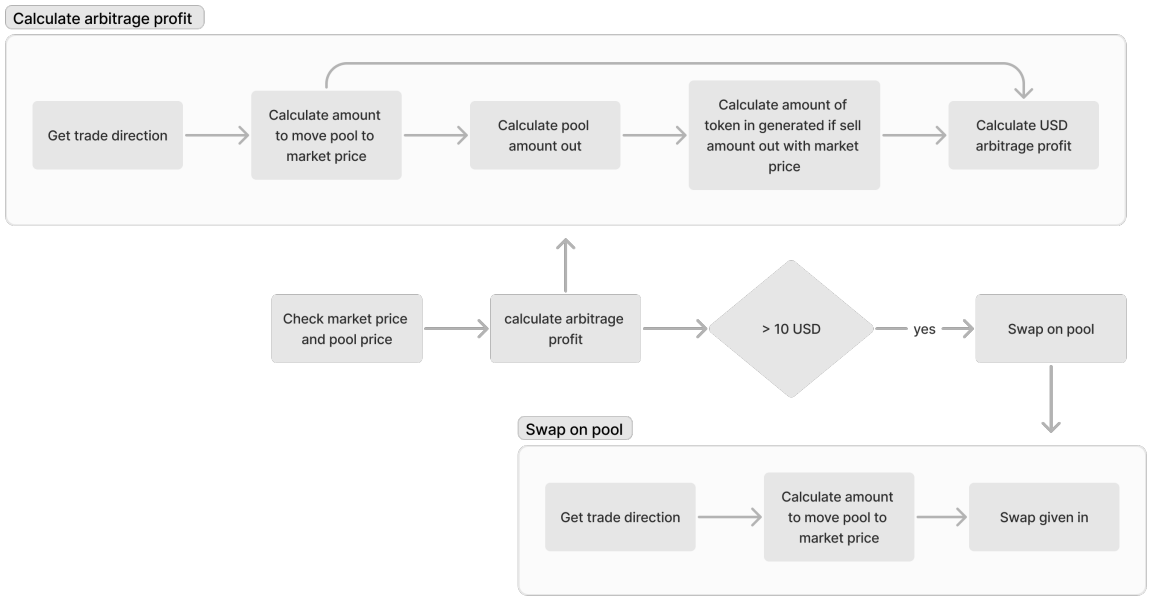
will assume that each join and exit action is perfectly balanced with the current state of the pool. Consequently, these join and exit events do not influence the pool's price. The update of the pool liquidity is represented by equation (6), where B is the balance of each pool token, T the current step of the simulation, and BPT the total pool shares.

$$B_{t+1} = B_t \left(1 + \frac{BPT_{t+1} - BPT_t}{BPT_t}\right) \tag{6}$$

3.4 ARBITRAGEUR

The arbitrageur agent in the simulation will attempt to extract value from the AMM at each step by updating its price. The arbitrageur will execute an arbitrage trade whenever it can secure at least 10 USD in profit. It is assumed that these trades will not impact the market price, as the market is sufficiently large to absorb the trades without price movement. The arbitrageur behavior is illustrated in Figure 13.

Figure 13 – Arbitrageur model action



Source: Author

To simplify the arbitrageur behavior, it was considered that the USDC token has a value equal to USD during the simulation period.

3.5 AGGREGATOR

The aggregator historical trades will be simulated within the pool to assess if the pool can generate a higher output amount compared to other routes. If the pool provides a superior output, it is assumed that the trade would be routed through the pool. For simplicity, all trades executed by the aggregator agent will be considered as sell orders with a fixed

input amount, ensuring consistency and comparability across different trading scenarios within the model. Partial executions of trades will not be implemented, since it would involve a bigger understanding of the routing algorithms of each DEX aggregator, increasing the model complexity.

3.6 VALIDATION

Two tests were conducted to validate the model. The first test involved simulating one week of activity using a constant swap fee to observe if the pool price would accurately follow the market price, mirroring the behavior of an actual pool. The second test comprised simulating 30 different weeks with the same historical pool swap fee, to evaluate if the fees generated by the model were comparable to the real fee collected.

3.6.1 Price test

The pool price validation was specifically designed to assess the effectiveness of the arbitrageur agent within the simulation. In this simulation, a constant fee, equal to 1%, was used to determine the boundaries within which arbitrage opportunities could occur. As displayed in Figure 14 the AMM price follows the market price which validates the arbitrageur agent design. This test is important since it proves that the funds of the LPs will be rebalanced over time.

Figure 14 – Simulation of AMM price with constant fee

Source: Author

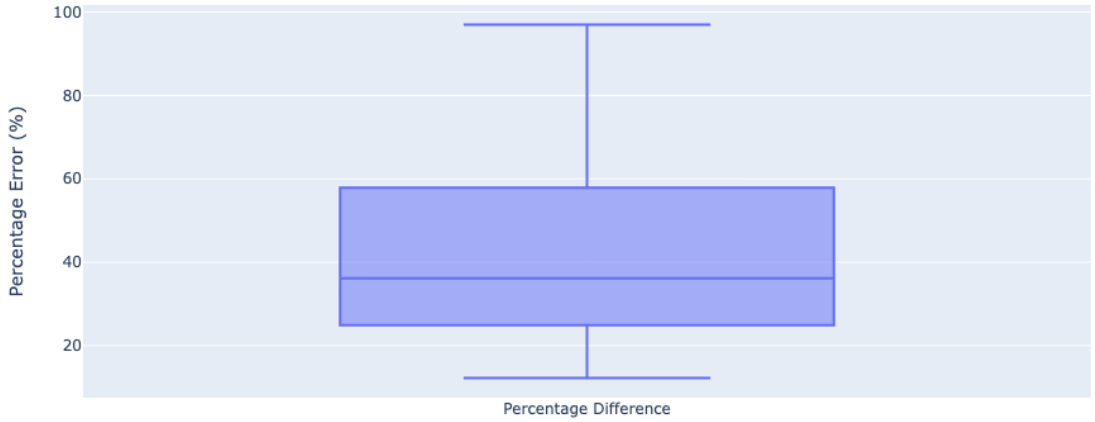
3.6.2 Fee collected test

The fee collection test will be applied to evaluate the model's ability to imitate the real market conditions. By using historical fee data from 30 different weeks of simulations and

comparing the simulation results with the actual historical fees collected, we can assess the accuracy and effectiveness of the model.

First, Figure 15 shows the distribution of the percentage absolute error over the 30 samples. In addition, the median percentage error is approximately 36%.

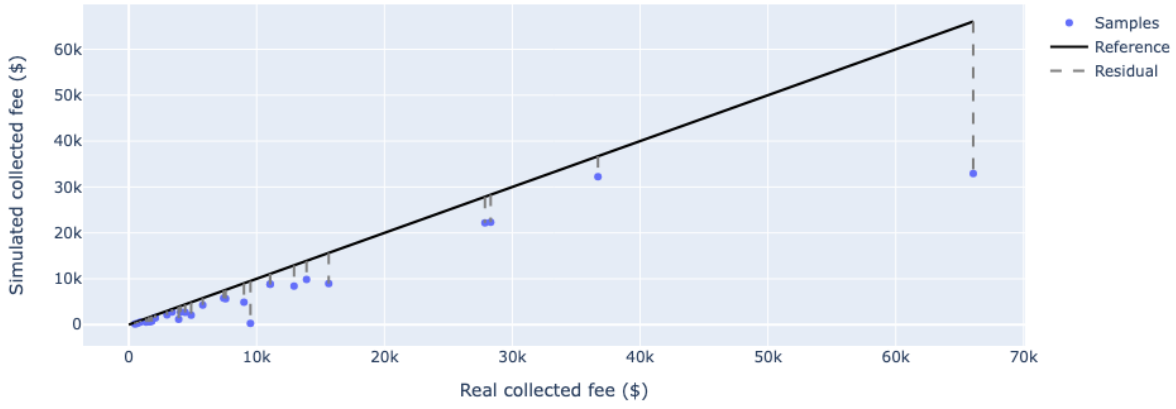
Figure 15 – Distribution of percentage error



Source: Author

Next, Figure 16 shows the relationship between the real and simulated collected fees. With that, it is possible to observe that the model generated a lower fee for all of the samples. This is expected since the simulated environment represents a part of the AMM market with just a subset of its components, which means fewer swaps and fees are collected.

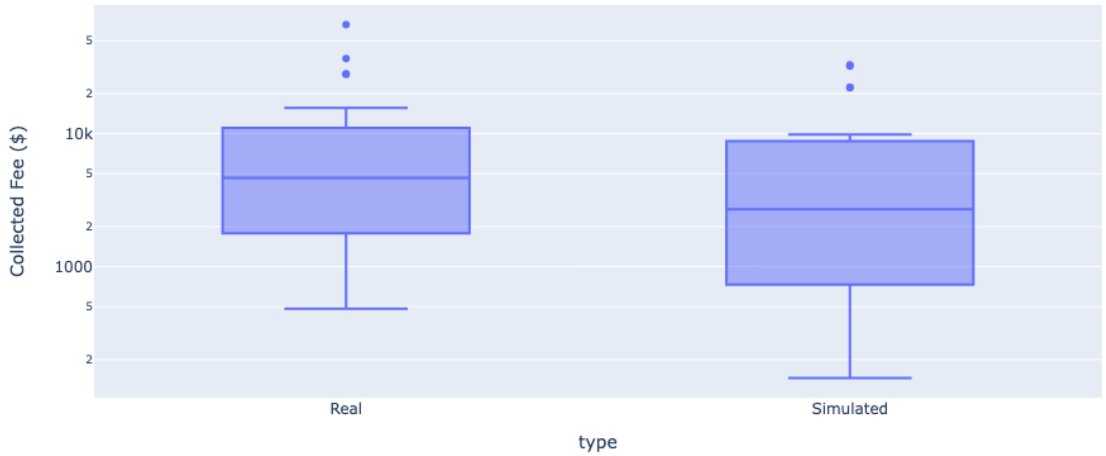
Figure 16 – Simulated and real collected fee samples



Source: Author

Lastly, the distribution of the real and simulated collected fee is represented in Figure 17. Here we can confirm the bias of the model to generate fewer fees collected.

Figure 17 – Distribution of real and simulated collected fee



Source: Author

It is crucial to notice the inherent complexity of the AMM market, making the development of a highly accurate simulation model a challenging task. The results of this study indicate that, despite modeling only two agents, the model can capture over 60% of the market's behavior. Consequently, this model provides a valid methodology for experimenting with new AMM mechanisms, even though some differences may occur in real market implementations.

4 AUTOMATED FEE MODEL ALGORITHMS

In this chapter, multiple automated fee model algorithms will be designed and tested. The fee models being developed have two primary objectives: to decrease LVR and to increase the regular trade volume. To achieve this, the idea is to apply higher fees on arbitrage trades and lower ones on regular trade activity.

However, the AMM has some data limitations, including the inability to directly determine whether a trade is arbitrage. On the blockchain, data such as the external market price would typically be accessible only through oracles. Pools that rely on price oracles introduce additional risk of exploitation since the oracles can be manipulated or subject to latency issues, leading to inaccurate pricing information (MUHS, 2023). For this reason and to decrease model complexity, only AMM internal data will be considered to design the fee models.

To ensure that the fees remain within reasonable limits, a saturation method will be applied to all models. The minimum fee will be set at 0.01%, while the maximum fee at 10%. This approach ensures that the fee adjustments remain effective and manageable, avoiding extreme values that could disrupt the trading environment or undermine the pool's stability.

Multiple parameters will be evaluated in a one-week simulation to quickly assess the viability and impact of the proposed models in a simplified method. In addition, the baseline used for evaluating the new fee models will be the historical fee used on the pool. This baseline provides a stable reference point against which the performance of the dynamic fee models can be measured. By comparing the outcomes of the new fee models to this constant fee, it will be possible to select the parameters of the new algorithms.

Three models will be designed, each implementing the same underlying principle: to adjust fees based on trading activity. In moments with high trade activity, the fee will be raised. Conversely, if trade activity is low, the fee will be lowered. While the core concept remains consistent across all three models, each will employ a different implementation.

Detailed explanations of each algorithm will be provided in the subsequent sections of this chapter, followed by comparative tests of all models against a constant fee benchmark to evaluate their effectiveness.

4.1 INCREMENTAL VOLUME ALGORITHM

The first algorithm will adjust fees based on the trading activity observed in the previous block. Specifically, the fee will decrease if no trades occur in the previous block, at the current rate. In addition, if a trade occurs, the fee is increased proportionally to the volume exchanged. This approach aims to dynamically balance the fee structure by responding to real-time trading volumes. In that sense, the math functions that determine the fee of each block are defined by equation (7), where F is the swap fee percentage, V is the volume swapped, and L the pool liquidity.

$$F_t = \begin{cases} F_{t-1} - \alpha_1 & , \ V_{t-1} = 0 \\ F_{t-1} + \alpha_2 \dfrac{V_{t-1}}{L_{t-1}} & , \ V_{t-1} > 0 \end{cases} \tag{7}$$

4.1.1 Parameter definition

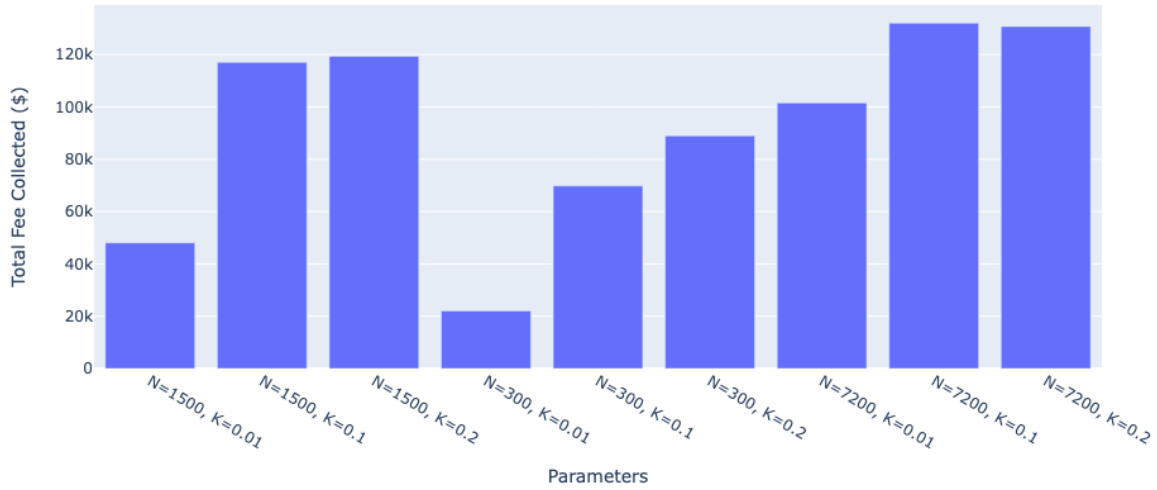
Several parameter options for α_1 and α_2 will be selected and simulated over a one-week period to determine the most effective configuration. For the selection of α_1 parameters, it was used the equation (8), where the N is the block number necessary to reset the fee from its maximum to the minimum value. The N values selected for the tests were 300, 1500, and 7200, corresponding approximately to the number of blocks generated over 1, 5, and 24 hours on the Ethereum blockchain.

$$\alpha_1 = \frac{F_{max} - F_{min}}{N} \tag{8}$$

For α_2, the aggregator's and snapshot data will be utilized to determine the median volume per liquidity value $\mu_{V/L}$. Subsequently, equation (9) will be applied, where K represents the increment of the fee range with the average trade. The chosen values for K are 0.2, 0.1, and 0.01.

$$\alpha_2 = \frac{F_{max} - F_{min}}{\mu_{V/L}} K \tag{9}$$

Figure 18 – Fee collected on one-week simulation for different parameters of the Incremental Volume Algorithm

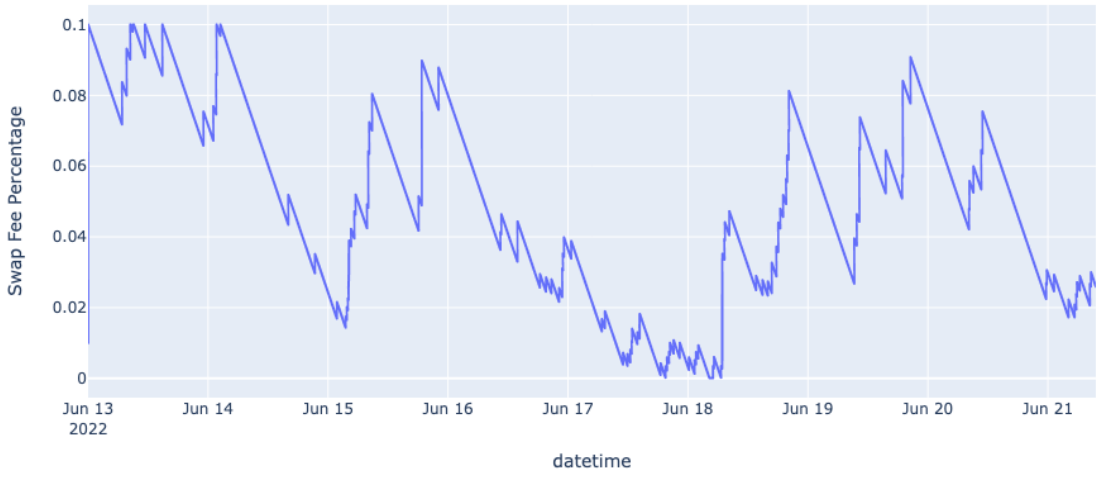


Source: Author

In this context, all parameter combinations were evaluated through a one-week simulation using identical historical data. Figure 18 presents the total collected fees for each simulation scenario. The results indicate that the optimal parameters are $(N, K) = (7200, 0.1)$.

It is also possible to analyze the swap fee percentage over time in Figure 19. In that context, the swap fee percentage increases fast in moments of big trades, taking up to 1 day to reset. This algorithm causes some instability for the pool price, which could be a problem in interacting with and integrating with aggregators, however, this aspect is not being considered in this work.

Figure 19 – Swap fee percentage over time on one-week simulation of the Incremental Volume Algorithm considering the best parameters



Source: Author

4.2 PRICE VOLATILITY ALGORITHM

The price volatility algorithm aims to dynamically adjust the transaction fees based on the observed volatility in the pool's price over recent blocks. The core idea behind this algorithm is to increase the fee if the pool's price exhibits significant fluctuations within a specified window of the previous blocks since this is a sign that arbitrageurs can extract value from the pool. By using the standard deviation of the price changes as a reference, the algorithm can identify periods of high volatility and respond accordingly by raising fees to mitigate risks and discourage potentially destabilizing trades.

This approach ensures that the fee structure remains responsive to market conditions, thereby enhancing the stability and efficiency of the liquidity pool. In that sense, the math function that defines this fee algorithm is shown in equation (10), where N is the number of blocks used, P the pool price, and β_1 is a parameter of the model.

$$F_t = \sigma(P_{t-N} : P_t)\beta_1 \tag{10}$$

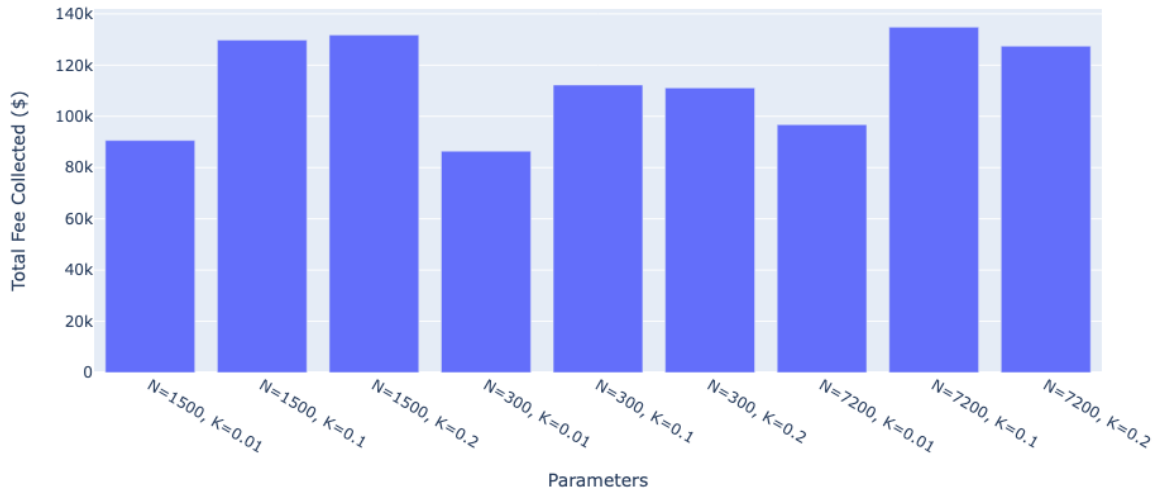
4.2.1 Parameter definition

For the N parameter, the same strategy of the section will be used. In addition, for the β_1 definition, we will use the median standard deviation of the ETH price on the Binance dataset (μ_σ) considering each window size. Then, β_1 will be defined using the equation (11), the chosen values for K were equal to those of the incremental volume algorithm.

$$\beta_1 = \frac{F_{max} - F_{min}}{\mu_\sigma} K \tag{11}$$

As was done on the other algorithm, all parameters were tested and the fee collected for each of them is illustrated in Figure 20 and the best result was with $(N, K) = (7200, 0.1)$.

Figure 20 – Fee collected on one-week simulation for different parameters of the Price Volatility Algorithm



Source: Author

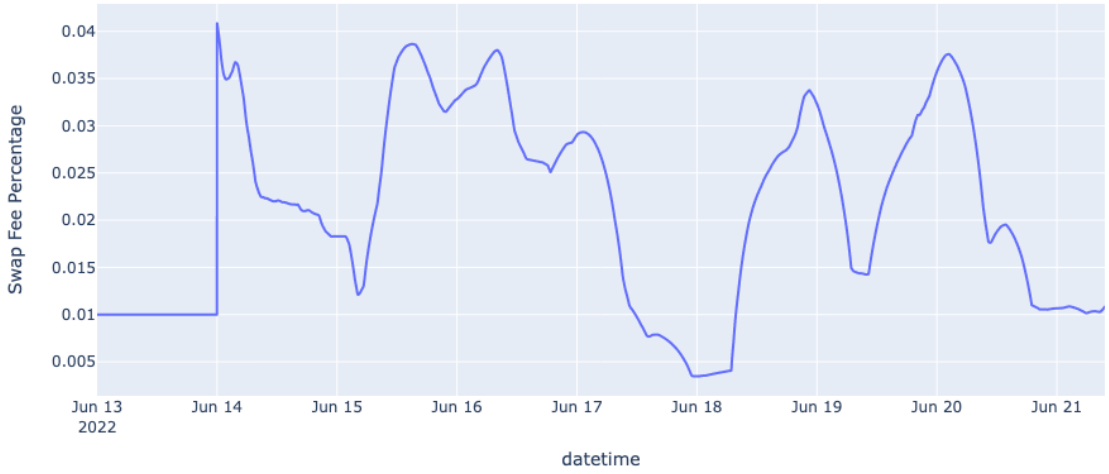
The swap fee applied in the simulation with the best parameters is represented in Figure 21. Here, it is possible to check that at the beginning of the period the fee is constant, this happens because of the window size.

Also, it is possible to check that this algorithm generates a smoother fee over time, this happens because, after a big trade, the swap fee will get to its maximum value after half of the window size.

4.3 VOLUME WINDOW ALGORITHM

The volume window volatility algorithm is designed to adjust transaction fees based on the trading volume within a specified window of recent blocks. By analyzing the volume of transactions over the previous N blocks, the algorithm dynamically responds to changes in market activity.

Figure 21 – Swap fee percentage over time on one-week simulation of the Price Volatility Algorithm considering the best parameters



Source: Author

This approach aims to balance the liquidity in the pool by discouraging excessive trading during periods of high volume, which can lead to LVR, and encouraging more trading when volumes are low. The math function that describes this algorithm is represented in equation (12), where μ represents the mean of the volume over the liquidity on the previous N blocks and φ_1 represents a gain to be set.

$$F_t = \sum_{i=t-N}^{t} \frac{V_i}{L_i} \varphi_1 \tag{12}$$

4.3.1 Parameter definition

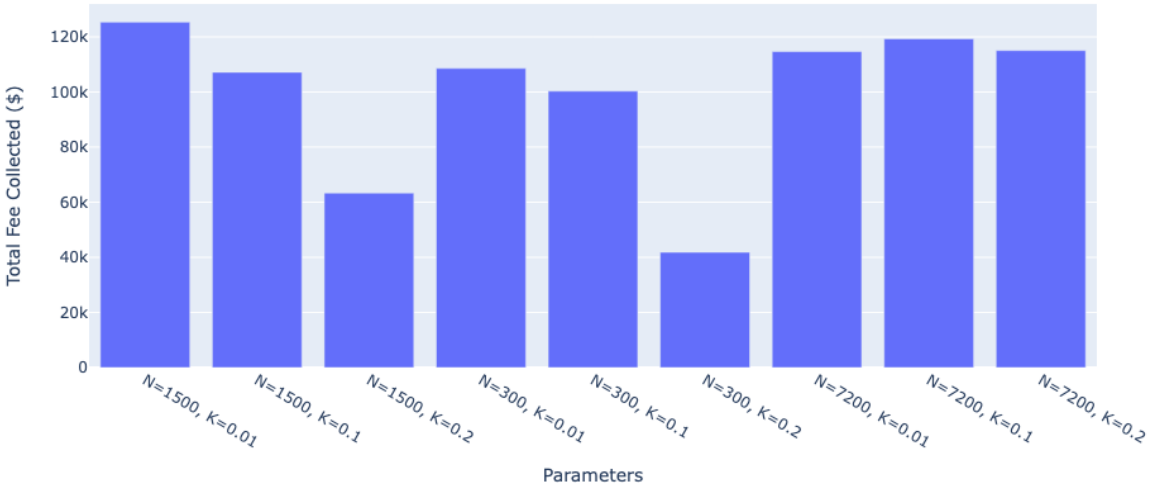
As was made on the last algorithm, the options for N will be the same. Also, the gain K will be used to set the φ_1 as represented in equation (13). The K options will be the same used on the other algorithms

$$\varphi_1 = \frac{F_{max} - F_{min}}{\mu_{V/L}} K \tag{13}$$

In Figure 22 the results in terms of the fee collected for each algorithm are represented. The best parameter set was the $(N, K) = (300, 0.01)$.

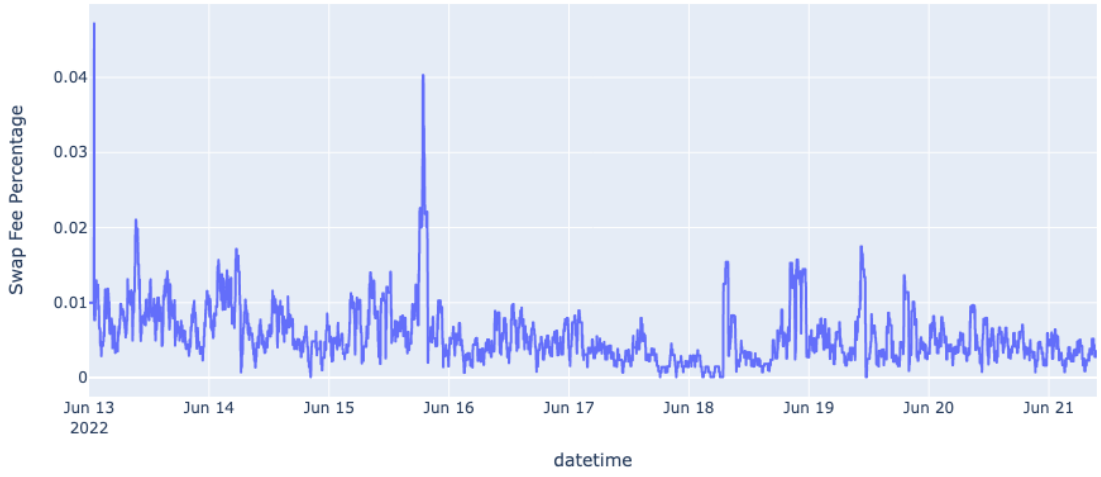
The swap fee percentage signal over time is illustrated in Figure 23. The swap fee signal exhibits some degree of noise, which can be attributed to the algorithm's behavior: the fee increases immediately following a trade and subsequently decreases after N blocks by an equivalent magnitude. This pattern reflects the algorithm's dynamic response to trading activity within the specified window.

Figure 22 – Fee collected on one-week simulation for different parameters of the Volume Window Algorithm



Source: Author

Figure 23 – Swap fee percentage over time on one-week simulation of the Volume Window Algorithm considering the best parameters



Source: Author

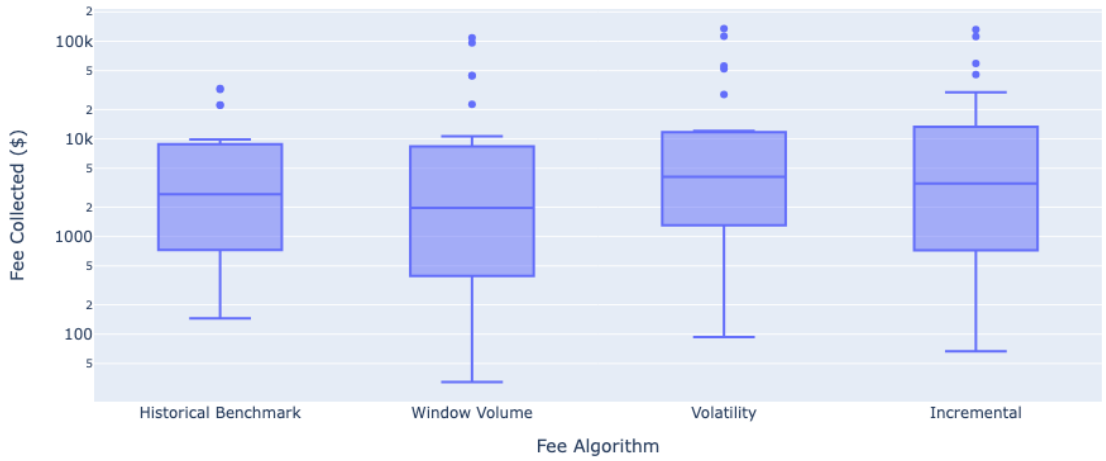
4.4 ALGORITHM TESTS

The three proposed algorithms will be tested over a span of 30 different weeks to evaluate their performance and effectiveness. For comparative purposes, the historical baseline fee algorithm will also be included in the testing process. The objective of this test is to find at least one algorithm that can overcome the historical benchmark.

This approach ensures that the dynamic fee models can be evaluated against a stable reference point, providing insights into how each algorithm adapts to varying market conditions and trading volumes. The results from these tests will help determine the most efficient fee adjustment strategy for optimizing liquidity provider returns.

Figure 24 presents a box plot illustrating the fees collected using each of the models. This visualization allows a comparative analysis of the distribution and variability of the fees generated by the three proposed dynamic fee algorithms, as well as the constant fee algorithm, over the span of 30 different weeks. Analyzing its results, the Incremental and Volatility algorithms outperformed the Historical benchmark in mean value, by $1,000 and $1,500.

Figure 24 – Distribution of collected fee for each tested fee model

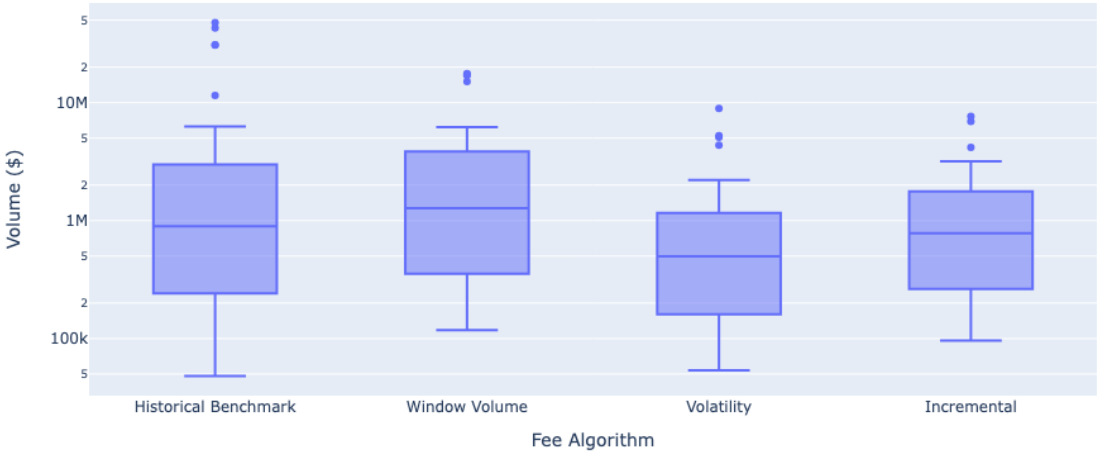


Source: Author

As a secondary metric, Figure 25 illustrates the volume routed by each algorithm on the tests. The Window and Historical Benchmark algorithms facilitated a higher volume of transactions during the tests, despite collecting lower fees. This outcome is attributed to their application of lower fees during the simulation, which encouraged a greater number of swaps. However, the lower fee collection suggests that these algorithms may have set the fees too low. In contrast, the Volatility algorithm outperformed the Incremental algorithm, with a median increase in fee collection of $300,000.

In summary, the tests indicate that the Volatility and Incremental algorithms are more profitable options for LPs as both outperformed the historical benchmark in terms of collected fees. While the Volatility algorithm was superior in fee collection, the Incremental algorithm

Figure 25 – Distribution of volume routed for each tested fee model



Source: Author

demonstrated a greater ability to attract swaps. In addition, the smart contract implementation of the Volatility algorithm includes more complex mathematical functions and additional variables. Taking these factors into account, the Incremental algorithm was chosen as the more appropriate candidate for implementation in the smart contract, as discussed in Chapter 5.

5 SMART CONTRACT IMPLEMENTATION

This chapter will address the smart contract implementation of a dynamic fee algorithm. Based on performance evaluations, the incremental volume algorithm was chosen over the other models. The implementation details will include the logic for adjusting fees based on recent trading volumes, ensuring the dynamic adjustment mechanism is accurately reflected within the contract's code.

The smart contract will be developed in the context of the Balancer protocol, utilizing its second version. Balancer V2 is particularly advantageous as it offers an interface that facilitates the integration of new mechanisms in custom pools.

5.1 POOL IMPLEMENTATION

In this section, the detailed implementation of the pool contract will be explained, showing the modifications made to integrate the incremental volume algorithm fee model. Following the pool implementation, the development of the pool factory will be addressed. Finally, the section will cover the tests conducted to ensure the functionality and reliability of the implemented features, demonstrating how the new dynamic fee model performs.

Gas cost optimization will not be the focus of this implementation. Instead, the primary goal is to demonstrate the functionality and feasibility of the automatic fee adjustment mechanism. As such, the contract will prioritize clarity and correctness over gas efficiency, providing a clear proof-of-concept rather than a production-ready solution.

Also, all this section code, including the tests, will be developed using Solidity language. Solidity is the most common language for smart contract development in the Ethereum ecosystem. (CHAINLINK, 2024)

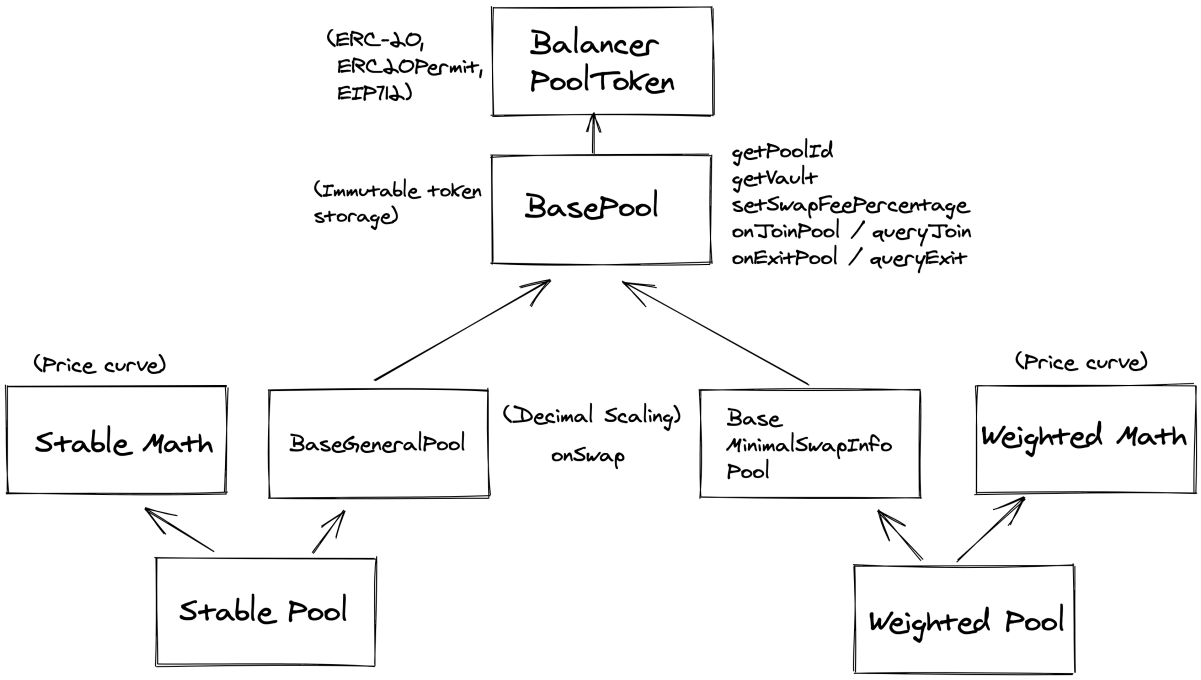
5.1.1 Pool with dynamic fee

The contract implementation will cover the incremental volume algorithm fee model for a pool containing two tokens. This AMM will resemble a weighted product model but with the new automatic fee adjustment mechanism based on trading activity. The algorithm will dynamically adjust the swap fees according to the trading activity, increasing fees during periods of high volume to protect liquidity providers and decreasing when there is no trade to encourage more activity.

The Balancer V2 built-in pools interface is illustrated in Figure 26. In that sense, it is possible to see that the support of custom pools goes beyond the vault centralization since some Base classes were built to facilitate the process. However, originally, the control of the swap fee is made by the *BasePool* contract. These methods will be overridden on the developed custom pool contract.

Most of the developed pool contract's AMM logic will closely follow the structure

Figure 26 – Simplified version of pools contracts relations



Source: (LABS, B., 2024)

of the weighted pool contract. To implement our incremental volume algorithm fee model, the weighted pool contract will be transformed to an abstract type to be extended. Then, the swap fee-relevant parts will be refactored. This adaptation will involve modifying the fee calculation to incorporate dynamic adjustments based on trading volume and ensuring that the initialization process aligns with the new fee model. By leveraging the established logic of the weighted pool contract, we can efficiently integrate our enhancements while maintaining the already-tested functionalities.

The smart contract will be explained through a combination of descriptive text and embedded code snippets, to provide a clear and detailed understanding of its implementation.

During the pool initialization, the constructor will call the contract parent method to ensure that the foundational elements of the pool are correctly set up. This includes initializing the standard parameters and configurations required for the pool to function as intended. In addition to these standard procedures, the constructor will also initialize the variables specific to the dynamic fee algorithm.

The swap fee is represented as an unsigned integer of 32 bytes with 18 decimals, meaning that a 100% fee is denoted as 10^{18}. Fixed-point math is utilized for fee calculations to handle the precision of decimal values. Safe math operations are used when dealing with block number information, as these values do not contain decimals. Both fixed-point math and safe math packages protect the code from overflow errors. In addition, rounding will always be implemented in favor of the pool as a security practice.

```
1 pragma solidity ^0.7.0;
2 pragma experimental ABIEncoderV2;
```

```
3
4  import "./AbstractWeightedPool.sol";
5  import "@balancer-labs/v2-solidity-utils/contracts/openzeppelin/SafeMath
       .sol";
6
7  contract WeightedPoolWithDynamicFee is AbstractWeightedPool {
8      using FixedPoint for uint256;
9
10     uint256 public lastSwapBlockNumber;
11     uint256 public lastSwapFeePercentage;
12     uint256 public minimalSwapFeePercentage;
13     uint256 public maximumSwapFeePercentage;
14     uint256 public feeDecayPerBlock;
15     uint256 public feeIncreasePerTradeSize;
16
17     struct DynamicFeeParams {
18         uint256 initialSwapFeePercentage;
19         uint256 minimalSwapFeePercentage;
20         uint256 maximumSwapFeePercentage;
21         uint256 feeDecayPerBlock;
22         uint256 feeIncreasePerTradeSize;
23     }
24
25     constructor(
26         AbstractWeightedPool.NewPoolParams memory params,
27         IVault vault,
28         IProtocolFeePercentagesProvider protocolFeeProvider,
29         uint256 pauseWindowDuration,
30         uint256 bufferPeriodDuration,
31         address owner,
32         DynamicFeeParams memory dynamicFeeParams
33     ) AbstractWeightedPool(params, vault, protocolFeeProvider,
           pauseWindowDuration, bufferPeriodDuration, owner) {
34         lastSwapBlockNumber = block.number;
35         lastSwapFeePercentage = dynamicFeeParams.
               initialSwapFeePercentage;
36         minimalSwapFeePercentage = dynamicFeeParams.
               minimalSwapFeePercentage;
37         maximumSwapFeePercentage = dynamicFeeParams.
               maximumSwapFeePercentage;
38         feeDecayPerBlock = dynamicFeeParams.feeDecayPerBlock;
39         feeIncreasePerTradeSize = dynamicFeeParams.
               feeIncreasePerTradeSize;
40     }
41
42     ...
```

Listing 5.1 – Pool initialization

Next, the getter and setter internal methods for the swap fee will be overridden from the parent contract to accommodate the dynamic fee adjustment mechanism. In the regular implementation, setting the swap fee involved emitting an event and reverting the transaction if the fee exceeded predefined boundaries. However, with the new dynamic fee model, the event emission has been removed because the swap fee is now updated at each swap. Instead of reverting, the swap fee is clipped to ensure it remains within the thresholds. In addition, the block number of the last block when the fee was updated is recorded.

For retrieving the swap fee, the getter method implements a decay parameter to compute the current fee. This is because the swap fee decays by a fixed amount with each block that passes without a trade. By using this decay parameter, the getter method can accurately reflect the current fee at any given block.

```
1    ...
2
3    function getSwapFeePercentage() public view override returns (
        uint256) {
4        uint256 blocksSinceLastSwap = block.number.sub(
            lastSwapBlockNumber);
5        uint256 feeDecay = SafeMath.mul(blocksSinceLastSwap,
            feeDecayPerBlock);
6
7        if (feeDecay >= lastSwapFeePercentage) {
8            return minimalSwapFeePercentage;
9        }
10
11       uint256 newSwapFee = lastSwapFeePercentage.sub(feeDecay);
12
13       if (minimalSwapFeePercentage > newSwapFee) {
14           return minimalSwapFeePercentage;
15       }
16       return lastSwapFeePercentage.sub(feeDecay);
17   }
18
19   function _setSwapFeePercentage(uint256 swapFeePercentage) internal
        virtual override {
20       if (swapFeePercentage < minimalSwapFeePercentage) {
21           swapFeePercentage = minimalSwapFeePercentage;
22       }
23
24       if (swapFeePercentage > maximumSwapFeePercentage) {
25           swapFeePercentage = maximumSwapFeePercentage;
26       }
27
28       lastSwapBlockNumber = block.number;
29       lastSwapFeePercentage = swapFeePercentage;
30   }
```

```
31
32      ...
```

Listing 5.2 – Get and set fee methods

A new internal function has been introduced to increase the swap fee with every new trade. In the simulation, the USD value of each trade was used to determine the fee adjustment. However, since the pool contract does not have access to external market prices, the implementation will instead use the token values for this calculation. This approach relies on the ratio between the amounts of the two tokens being traded, which remains the same independent of its unit. Another change added was that this implementation will be called every swap, instead of each block.

```
1       ...
2
3       function _setSwapFeeAfterSwap(uint256 currentBalance, uint256
            tradeAmount) internal {
4           uint256 tradeSize = tradeAmount.divUp(currentBalance);
5           uint256 swapFeeIncrease = tradeSize.mulUp(
                feeIncreasePerTradeSize);
6           setSwapFeePercentage(swapFeeIncrease.add(getSwapFeePercentage())
                );
7       }
8
9       ...
```

Listing 5.3 – Fee setting after each swap

The internal functions for handling token swaps considering fixed input or output will be extended to include a call to the swap fee setting function after each swap is executed. This extension ensures that the swap fee is dynamically adjusted based on the trade information.

```
1       ...
2
3       function _onSwapGivenIn(
4           SwapRequest memory swapRequest,
5           uint256 currentBalanceTokenIn,
6           uint256 currentBalanceTokenOut
7       ) internal override returns (uint256) {
8           uint256 outGivenIn = super._onSwapGivenIn(swapRequest,
                currentBalanceTokenIn, currentBalanceTokenOut);
9           _setSwapFeeAfterSwap(currentBalanceTokenIn, swapRequest.amount);
10          return outGivenIn;
11      }
12
13      function _onSwapGivenOut(
14          SwapRequest memory swapRequest,
15          uint256 currentBalanceTokenIn,
16          uint256 currentBalanceTokenOut
```

```
17      ) internal virtual override returns (uint256) {
18          uint256 outGivenIn = super._onSwapGivenOut(swapRequest,
                currentBalanceTokenIn, currentBalanceTokenOut);
19          _setSwapFeeAfterSwap(currentBalanceTokenIn, swapRequest.amount);
20          return outGivenIn;
21      }
22  }
```

Listing 5.4 – On swap internal methods

5.1.2 Pool factory

In the context of smart contracts, a factory is a specialized contract designed to create and manage instances of other contracts, often referred to as child contracts. A pool contract factory is a valuable tool in the development and deployment of DeFi applications because it facilitates the creation and management of multiple liquidity pools. By using a factory contract, developers can standardize the deployment process, ensuring that each new pool has a consistent set of parameters and configurations. This not only reduces the potential for errors but also simplifies the process of pool creation, making it more efficient and scalable.

The pool factory implementation uses the base Pool Factory interface provided by Balancer. This interface simplifies the creation of new pools by requiring only the specific parameters for the pool and the dynamic swap fee settings. Core parameters such as the Vault are pre-configured within the factory. The contract can be observed in the annex A.

5.1.3 Automated tests

The fee behavior of the functions will be tested using automated tests with Forge Solidity. These tests will focus exclusively on the new dynamic fee methods, as the rest of the contract relies on the already-tested and audited Balancer Weighted pool contract. Automated tests are crucial in smart contract development because once the contract is deployed it can not be updated. In that sense, if there is some vulnerability that can be exploited, user funds might be at risk and beyond the protection of the protocol due to the trustless characteristic of the implementation.

The setup of the automated tests started with the deployment of a mocked version of an authorizer contract, which is necessary for the Vault's operations. Following this, the Vault, Protocol Fee collector, and the pool factory were deployed. A new pool was then created using two mocked versions of ERC-20 tokens, using two equal weights tokens as in the simulation. The fee-related parameters were chosen to facilitate the development and evaluation of the dynamic fee adjustment mechanism. In addition, a join was made to provide liquidity on the pool. With that setup, the tests were divided into 6 parts and can also be verified in the annex B.

- **Swap fee do not decay in the same block:** It asserts that, if the last block where the swap fee was updated matches the current block, the last recorded swap fee percentage is equal to the current swap fee percentage;
- **Swap fee decay over blocks**: Starting with the initial swap fee percentage, it simulates the progress of 100 blocks and then checks that the swap fee has decreased. It also confirms that the decrease is the expected value;
- **Swap fee does not drop below minimum**: By rolling the block number forward by a large amount (1 million), it checks that the swap fee is equal to the minimal swap fee defined;
- **Updated swap fee is used on swaps**: After rolling a significant amount of blocks, it calculates the expected output amount using the decayed swap fee and confirms that the resulting token balance matches this expected amount;
- **Swap fee increases after swap**: It starts by recording the initial swap fee, performs a swap, and then verifies that the swap fee has increased appropriately based on the trade size and the predefined parameters;
- **Swap fee not overcome maximum**: It performs a swap with a large input amount and then checks that the swap fee is equal to the maximum swap fee percentage.

5.1.4 Gas consumption

Even though the code was not specifically developed with a primary focus on gas optimization, gas usage was still evaluated for a single swap operation in both the weighted pool and the newly developed dynamic fee model. This evaluation helps to understand the gas cost implications of the dynamic fee adjustments compared to the standard weighted pool operations. Table 5 shows the deployment size on both swaps calculated with Forge solidity package.

Contract	Swap call deployment size (bytes)
WeightedPool	111052
WeightedPoolWithDynamicFee	138172

Table 5 – Swap gas report

6 CONCLUSION

6.1 CONCLUSIVE SUMMARY

The primary goal of this thesis was to propose a methodology for evaluating novel pool mechanisms within AMMs, specifically focusing on finding at least one automatic fee model that outperforms the traditional constant fee structure and validating the feasibility of its implementation. The research aimed to identify a dynamic fee adjustment mechanism that could improve LP returns.

The research concluded that there is significant potential for optimization in the fee management of liquidity pools. It was observed that LPs often lose money due to LVR, and the behavior of key agents such as aggregators and arbitrageurs plays a crucial role in this dynamic.

The simulation results demonstrated that, even though the DeFi market is extremely complex and requires careful consideration, 64% of the total fees collected by the selected liquidity pool were achieved by simulating aggregator and arbitrageur behavior. Historical data were crucial to validate the simulation methodology.

In the fee models testing phase, three different automatic fee models were developed and compared against a constant fee baseline. Two of these models outperformed the baseline, showing 33% increase in total fee collected in the simulated scenario. Among them, the simplest model was selected for further development due to its effectiveness and ease of implementation.

The selected fee model was successfully integrated and tested within the Balancer V2 protocol. Automated tests validated its functionality, confirming that the dynamic fee adjustments worked as intended. Despite the added complexity, the gas costs for this model were about 25% higher than those for the standard weighted pool.

The main objectives of the thesis were accomplished. A robust methodology for testing fee mechanisms was developed, and an automatic fee model was identified that outperforms the constant fee approach. The feasibility of implementing this model within existing AMM frameworks was demonstrated, providing a viable tool to combat LVR.

6.2 FUTURE WORK

This section will outline the proposed improvements for the project, focusing on three key areas: the simulation methodology, the fee model algorithms, and the smart contract implementation. Enhancements in the simulation methodology have the potential to increase the accuracy and realism of market behavior predictions. In that sense, improvements to the fee model algorithms can also optimize fee adjustments for better liquidity provider returns. Lastly, adjustments in the smart contract implementation will target increased efficiency and security.

6.2.1 Simulation methodology improvements

Improvements to the simulation methodology will involve expanding the scope to include a broader range of assets. By simulating pools with different asset classes, it is possible to gain deeper insights into the dynamics of different types of liquidity pools.

Additionally, the simulation can be enhanced by introducing new agents, such as other pools that arbitrageurs can exploit for profit opportunities. This will provide a more comprehensive understanding of the interactions between various pools and the strategies employed by arbitrageurs, leading to more robust and realistic evaluations of the proposed fee models.

6.2.2 Modification of fee algorithm for different invariants

Another improvement would be to test different fee models for different pool invariants, checking what kind of fee algorithm provide better results for each AMM type. By adjusting the algorithm to work with different invariant curves, such as stable math or with liquidity concentrated, it is also possible to check if the proposed fee model is versatile and applicable to a wider range of liquidity pool structures.

6.2.3 Smart contract modifications

Modifying the smart contract to improve gas efficiency is extremely relevant, as high gas costs during swaps can significantly decrease trading activity, leading to reduced overall transaction volume and fees collected by LPs. By optimizing the smart contract to minimize gas consumption, we can enhance the user experience, making it more cost-effective for traders to interact with the pool. This optimization will involve refining the fee calculation algorithms and variables used.

In addition to gas efficiency, ensuring the security of the smart contract is essential for launching to the public. This involves conducting thorough audits and stress tests to identify and rectify potential vulnerabilities. An external audit by a reputable security firm will ensure that the contract adheres to the highest security standards, mitigating the risk of exploits and ensuring the safety of user funds. Also, stress tests can be developed to simulate various high-load and edge-case scenarios to verify the contract's robustness under different conditions.

6.2.3.1 Balancer V3

Balancer announced a new version of its protocol, which promises enhanced capabilities that could facilitate the implementation of the automatic fee mechanism more efficiently (X, 2023). Similar to the hooks introduced in Uniswap v4, these new features in Balancer's latest protocol version are expected to offer greater flexibility and modularity, making it easier to integrate dynamic fee adjustments directly into the pool logic (LABS, U., 2023).

In addition, the new protocol version focuses on improved gas efficiency, which would mitigate the higher gas costs associated with more complex fee algorithms. These advancements will not only simplify the integration process but also enhance the overall performance and cost-effectiveness of the automatic fee mechanism within the Balancer protocol. With this new version of the protocol, the developed smart contract will need to be modified to adapt to the newly proposed interface.

REFERENCES

ADAMS, Hayden; ZINSMEISTER, Noah; ZINSMEISTER, Noah. **Uniswap v2 Core**. 2024. Available from: https://uniswap.org/whitepaper.pdf.

BALANCER. **B-50USDC-50WETH Page**. 2021. Available from: https://app.balancer.fi/#/ethereum/pool/0x96646936b91d6b9d7d0c47c496afbf3d6ec7b6f8000200000000000000000019.

BINANCE. **What Is Wrapped Ether (WETH) and How to Wrap It?** 2022. Available from: https://academy.binance.com/en/articles/what-is-wrapped-ether-weth-and-how-to-wrap-it.

BUTERIN, Vitalik. **Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.** 2014. Available from: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.

CENTIC. **Centralized Exchange (CEX) vs Decentralized Exchange (DEX). Which is the best crypto exchange?** 2023. Available from: https://medium.com/@centicio/centralized-exchange-cex-vs-decentralized-exchange-dex-which-is-the-best-crypto-exchange-148f48ea51c1.

CENTRE. **USD Coin Whiteppaper**. 2018. Available from: https://whitepaper.io/document/716/usd-coin-whitepaper.

CHAINLINK. **Top 6 Smart Contract Languages in 2024**. 2024. Available from: https://chain.link/education-hub/smart-contract-programming-languages.

COINBASE. **What is a stablecoin?** 2024. Available from: https://www.coinbase.com/en-br/learn/crypto-basics/what-is-a-stablecoin.

COINMARKETCAP. **Glossary: Dex Aggregator**. 2024. Available from: https://coinmarketcap.com/academy/glossary/dex-aggregator.

DAO, Balancer Grants. **Bleu is building Balancer tooling, Twitter bots, simulators and much more!** 2023. Available from: https://medium.com/@BalancerGrants/bleu-is-building-balancer-tooling-twitter-bots-simulators-and-much-more-545b579c46e0.

E, Atis. **Dynamic Fees for Automated Market Makers: Liquidity, Volatility, and Collected Fees**. 2023. Available from: https://atise.medium.com/dynamic-fees-for-automated-market-makers-liquidity-volatility-and-collected-fees-db211da18d0d.

FRITSCH, Robin; CANIDIO, Andrea. **Measuring Arbitrage Losses and Profitability of AMM Liquidity**. [S.l.: s.n.], 2024. arXiv: 2404.05803 [cs.DC].

GATEWAY.FM. **Dynamic Fees for Automated Market Makers: Liquidity, Volatility, and Collected Fees**. 2023. Available from: https://uxplanet.org/the-state-of-web3-in-2023-cc2b5ae6f172.

GRISHCHENKO, Ilya; MAFFEI, Matteo; SCHNEIDEWIND, Clara. A Semantic Framework for the Security Analysis of Ethereum smart contracts. **CoRR**, abs/1802.08660, 2018. arXiv: 1802.08660. Available from: http://arxiv.org/abs/1802.08660.

HABER STUART STORNETTA, W. Scott. How to time-stamp a digital document. **Journal of Cryptology**, v. 20, p. 99–111, 1991. DOI: 10.1007/BF00196791.

JONES, Kevin. **Easily Access Blockchain Data with Subgraphs**. 2023. Available from: https://graphql.org/conf/2023/sessions/ebee6213b39b87437eb7cc9c41ea972b/.

KOZLOVSKI, Stanislav. **Balancer V2 — A One-Stop-Shop**. 2021. Available from: https://medium.com/balancer-protocol/balancer-v2-a-one-stop-shop-6af1678003f7.

KREITENWEIS, Angela. **Understanding Balancer Pools**. 2021. Available from: https://medium.com/balancer-protocol/developers-balancer-v2-smart-contracts-are-now-live-e97002ee0310.

KRYPTOMAT. **Types of Cryptocurrency**. 2024. Available from: https://kriptomat.io/cryptocurrencies/types-of-cryptocurrency/.

LABS, Balancer. **Balancer Docs**. 2024. Available from: https://docs.balancer.fi/concepts/overview/basics.html.

LABS, Balancer. **What Is Balancer? The Complete Guide**. 2021. Available from: https://medium.com/balancer-protocol/what-is-balancer-the-complete-guide-762ee230a9d4.

LABS, Cata. **Optimising LP Performance Part 2: Dynamic Fees**. 2023. Available from: https://blog.catalyst.exchange/optimising-lp-performance-part-2-dynamic-fees/.

LABS, Uniswap. **Uniswap Docs**. 2023. Available from: https://docs.uniswap.org/contracts/v4/overview.

MARTINELLI, Fernando. **Balancer Partners with Gauntlet To Make Dynamic-Fee Pools A Reality**. 2021a. Available from: https://medium.com/balancer-protocol/balancer-partners-with-gauntlet-to-make-dynamic-fee-pools-a-reality-97b3fb1760df.

MARTINELLI, Fernando. **Introducing Balancer V2: Generalized AMMs**. 2021b. Available from: https://medium.com/balancer-protocol/balancer-v2-generalizing-amms-16343c4563ff.

MILIONIS, Jason; MOALLEMI, Ciamac C.; ROUGHGARDEN, Tim; ZHANG, Anthony Lee. **Automated Market Making and Loss-Versus-Rebalancing**. [S.l.: s.n.], 2024. arXiv: 2208.06046 [q-fin.MF].

MUHS, Dominik. **Oracle Manipulation Attacks**. 2023. Available from: https://scsfg.io/hackers/oracle-manipulation/.

NAKAMOTO, Satoshi. **Bitcoin: A Peer-to-Peer Electronic Cash System**. 2008. Available from: https://bitcoin.org/bitcoin.pdf.

REGNATH, Emanuel; STEINHORST, Sebastian. SmaCoNat: Smart Contracts in Natural Language, p. 5–16, Sept. 2018. DOI: 10.1109/FDL.2018.8524068.

RODRIGUEZ, Miguel. **Cryptocurrency Arbitrage**. 2023. Available from: https://capex.com/de/akademie/crypto-arbitrage.

SCHUEFFEL, Patrick. DeFi: Decentralized Finance - An Introductionand Overview, p. i–xi, 2021. DOI: 10.24840/2183-0606_009.003_0001.

SERGEENKOV, Andrey. **What Are Sandwich Attacks in DeFi — and How Can You Avoid Them?** 2021. Available from: https://coinmarketcap.com/academy/article/what-are-sandwich-attacks-in-defi-and-how-can-you-avoid-them.

SIEGEL, David. **Understanding The DAO Attack**. 2016. Available from: https://www.coindesk.com/learn/understanding-the-dao-attack/.

SIMPLELEARN. **What is Solidity Programming: Data Types, Smart Contracts, and EVM?** 2023. Available from: https://www.simplilearn.com/tutorials/blockchain-tutorial/what-is-solidity-programming.

SOLARCURVE. **[BIP-295] Fee Discount for CowSwap Solvers**. 2023. Available from: https://forum.balancer.fi/t/bip-295-fee-discount-for-cowswap-solvers/4785.

TEAM, Dune Analytics'. **Dune Analytics Documentation**. 2024. Available from: https://docs.dune.com/home.

TEAM, Investopedia. **Gas (Ethereum): How Gas Fees Work on the Ethereum Blockchain**. 2024.

UNISWAP. **How Uniswap works**. 2020. Available from: https://docs.uniswap.org/contracts/v2/concepts/protocol-overview/how-uniswap-works.

VOGELSTELLER, Fabian; BUTERIN, Vitalik. **ERC-20: Token Standard**. 2015. Available from: https://eips.ethereum.org/EIPS/eip-20.

X, Beethoven. **Balancer V3: Embracing the Future of DeFi**. 2023. Available from: https://medium.com/balancer-protocol/balancer-v3-embracing-the-future-of-defi-46f37f4368aa.

XU, Jiahua; PARUCH, Krzysztof; COUSAERT, Simon; FENG, Yebo. SoK: Decentralized Exchanges (DEX) with Automated Market Maker (AMM) Protocols. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 55, n. 11, Feb. 2023. ISSN 0360-0300. DOI: 10.1145/3570639. Available from: https://doi.org/10.1145/3570639.

ZHOU H MILANI FARD A, Makanju A. The State of Ethereum Smart Contracts Security: Vulnerabilities, Countermeasures, and Tool Support. **Journal of Cybersecurity and Privacy**, p. 358–378, 2022. DOI: 10.3390/jcp2020019.

ANNEX A – POOL FACTORY CONTRACT CODE

```
pragma solidity ^0.7.0;
pragma experimental ABIEncoderV2;

import "@balancer-labs/v2-interfaces/contracts/vault/IVault.sol";

import "@balancer-labs/v2-pool-utils/contracts/factories/BasePoolFactory.sol";

import "./AbstractWeightedPool.sol";
import "./WeightedPoolWithDynamicFeeFromAbstract.sol";

contract WeightedPoolWithDynamicFeeFactory is BasePoolFactory {
    constructor(
        IVault vault,
        IProtocolFeePercentagesProvider protocolFeeProvider,
        uint256 initialPauseWindowDuration,
        uint256 bufferPeriodDuration
    )
        BasePoolFactory(
            vault,
            protocolFeeProvider,
            initialPauseWindowDuration,
            bufferPeriodDuration,
            type(WeightedPoolWithDynamicFee).creationCode
        )
    {
        // solhint-disable-previous-line no-empty-blocks
    }

    /**
     * @dev Deploys a new `WeightedPoolWithDynamicFee`.
     */
    function create(
        string memory name,
        string memory symbol,
        IERC20[] memory tokens,
        uint256[] memory normalizedWeights,
        IRateProvider[] memory rateProviders,
        address owner,
        bytes32 salt,
        WeightedPoolWithDynamicFee.DynamicFeeParams memory
            dynamicFeeParams
    ) external returns (address) {
        (uint256 pauseWindowDuration, uint256 bufferPeriodDuration) =
            getPauseConfiguration();
```

```
43
44          return
45              _create(
46                  abi.encode(
47                      AbstractWeightedPool.NewPoolParams({
48                          name: name,
49                          symbol: symbol,
50                          tokens: tokens,
51                          normalizedWeights: normalizedWeights,
52                          rateProviders: rateProviders,
53                          assetManagers: new address[](tokens.length), //
                                Don't allow asset managers,
54                          swapFeePercentage: 0 // Swap fee is dynamic
55                      }),
56                      getVault(),
57                      getProtocolFeePercentagesProvider(),
58                      pauseWindowDuration,
59                      bufferPeriodDuration,
60                      owner,
61                      dynamicFeeParams
62                  ),
63                  salt
64              );
65      }
66  }
```

Listing A.1 – Pool factory contract

ANNEX B – POOL TESTS CODE

```
pragma solidity ^0.7.0;
pragma experimental ABIEncoderV2;

import { Test } from "forge-std/Test.sol";
import "@balancer-labs/v2-solidity-utils/contracts/math/FixedPoint.sol";

import "../WeightedPoolWithDynamicFeeFromAbstract.sol";
import "../WeightedDynamicPoolFactory.sol";
import "../AbstractWeightedPool.sol";
import "../BaseWeightedPool.sol";

import { MockBasicAuthorizer } from "@balancer-labs/v2-solidity-utils/
    contracts/test/MockBasicAuthorizer.sol";
import "@balancer-labs/v2-solidity-utils/contracts/openzeppelin/ERC20.
    sol";
import "@balancer-labs/v2-standalone-utils/contracts/
    ProtocolFeePercentagesProvider.sol";
import "@balancer-labs/v2-solidity-utils/contracts/helpers/ERC20Helpers.
    sol";
import "@balancer-labs/v2-solidity-utils/contracts/openzeppelin/SafeMath
    .sol";
import "@balancer-labs/v2-interfaces/contracts/vault/IPoolSwapStructs.
    sol";
import "@balancer-labs/v2-interfaces/contracts/solidity-utils/
    openzeppelin/IERC20.sol";
import "../../vault/contracts/Vault.sol";

contract MyToken is ERC20 {
    constructor(string memory _name, string memory _symbol) ERC20(_name,
        _symbol) {}

    function mint(address to, uint256 amount) public virtual {
        _mint(to, amount);
    }

    function burn(address form, uint256 amount) public virtual {
        _burn(form, amount);
    }
}

contract WeightedPoolWithDynamicFeeFromFactoryTest is Test {
    using FixedPoint for uint256;

    MyToken private _token0;
    MyToken private _token1;
```

```
38
39      WeightedPoolWithDynamicFee private _pool;
40      IVault private _vault;
41      MockBasicAuthorizer private _authorizer;
42      ProtocolFeePercentagesProvider private _protocolFeeProvider;
43      WeightedPoolWithDynamicFeeFactory private _poolFactory;
44
45      uint256 private _currentBlock = 10000;
46
47      function setUp() public {
48          // mock tokens
49          _token0 = new MyToken("tokenA", "TKA");
50          _token1 = new MyToken("tokenB", "TKB");
51
52          // mock authorizer, vault, and protocol fee provider
53          _authorizer = new MockBasicAuthorizer();
54          _authorizer.grantRole(0x00, address(this));
55          _vault = new Vault(_authorizer, IWETH(0), 0, 0);
56          _protocolFeeProvider = new ProtocolFeePercentagesProvider(_vault
                , 1e18, 1e18);
57
58          // create pool factory
59          _poolFactory = new WeightedPoolWithDynamicFeeFactory(_vault,
                _protocolFeeProvider, 0, 0);
60
61          // grant swap role to vault
62          _authorizer.grantRole(0
                x60225c28c48f27009c1a16044a6be1035b625fedb3fab23b42f54140e9a83e8a
                , address(_vault));
63
64          // create pool parameters
65          IERC20[] memory tokens = _sortTokens(_token0, _token1);
66          uint256[] memory nomalizedWeights = new uint256[](2);
67          nomalizedWeights[0] = 50e16;
68          nomalizedWeights[1] = 50e16;
69          IRateProvider[] memory rateProviders = new IRateProvider[](2);
70          rateProviders[0] = IRateProvider(0);
71          rateProviders[1] = IRateProvider(0);
72          bytes32 salt = bytes32(0);
73
74          // update block number and timestamp
75          vm.roll(_currentBlock);
76          vm.warp(1000);
77
78          // create pool
79          address _poolAddress = _poolFactory.create(
80              "Test Pool",
```

```
81              "TEST",
82              tokens,
83              nomalizedWeights,
84              rateProviders,
85              address(this),
86              salt,
87              WeightedPoolWithDynamicFee.DynamicFeeParams({
88                  initialSwapFeePercentage: 1e15,
89                  minimalSwapFeePercentage: 1e14,
90                  maximumSwapFeePercentage: 1e17,
91                  feeDecayPerBlock: 1e10,
92                  feeIncreasePerTradeSize: 4e18
93              })
94          );
95          _pool = WeightedPoolWithDynamicFee(_poolAddress);
96
97          // mint tokens and join pool
98          _token0.mint(address(this), 1e21);
99          _token1.mint(address(this), 1e21);
100         _token0.approve(address(_vault), 1e21);
101         _token1.approve(address(_vault), 1e21);
102         uint256[] memory amounts = new uint256[](2);
103         amounts[0] = 1e21;
104         amounts[1] = 1e21;
105         bytes32 poolId = _pool.getPoolId();
106         _vault.joinPool(
107             _pool.getPoolId(),
108             address(this),
109             address(this),
110             IVault.JoinPoolRequest({
111                 assets: _asIAsset(tokens),
112                 maxAmountsIn: amounts,
113                 userData: abi.encode(WeightedPoolUserData.JoinKind.INIT,
                        amounts, 1e18),
114                 fromInternalBalance: false
115             })
116         );
117
118         // mint tokens and approve vault for test swaps
119         _token0.mint(address(this), 1e21);
120         _token0.approve(address(_vault), 1e21);
121     }
122
123     function testSwapFeeInTheSameBlock() public {
124         assertEq(_currentBlock, _pool.lastSwapBlockNumber());
125
126         assertEq(_pool.lastSwapFeePercentage(), _pool.
```

```
            getSwapFeePercentage());
    }

    function testSwapFeeDecayOverBlocks() public {
        uint256 startingFee = _pool.getSwapFeePercentage();
        vm.roll(_currentBlock + 100);
        uint256 feeAfter100Block = _pool.getSwapFeePercentage();
        assertLt(feeAfter100Block, startingFee);
        assertEq(feeAfter100Block.add((100 * _pool.feeDecayPerBlock())),
            startingFee);
    }

    function testSwapFeeNotGoesBelowMinimal() public {
        vm.roll(_currentBlock + 1e6);
        uint256 feeAfter1000Blocks = _pool.getSwapFeePercentage();
        assertEq(feeAfter1000Blocks, _pool.minimalSwapFeePercentage());
    }

    event Transfer(address indexed from, address indexed to, uint256
       amount);

    function testSwapFeeIsUsedOnSwap() public {
        vm.roll(_currentBlock + 1e6);
        uint256 startingFee = _pool.getSwapFeePercentage();
        uint256 amountIn = 1e15;
        uint256 amountInAfterSwapFee = amountIn.sub(amountIn.mulUp(
           startingFee));
        uint256 amountOut = WeightedMath._calcOutGivenIn(1e21, 50e16, 1
           e21, 50e16, amountInAfterSwapFee);

        _vault.swap(
            IVault.SingleSwap({
                poolId: _pool.getPoolId(),
                kind: IVault.SwapKind.GIVEN_IN,
                assetIn: IAsset(address(_token0)),
                assetOut: IAsset(address(_token1)),
                amount: amountIn,
                userData: new bytes(0)
            }),
            IVault.FundManagement({
                sender: address(this),
                recipient: payable(address(this)),
                fromInternalBalance: false,
                toInternalBalance: false
            }),
            0,
            1e8
```

```
169         );
170         assertEq(_token1.balanceOf(address(this)), amountOut);
171     }
172 
173     function testSwapFeeIncreaseOnSwap() public {
174         vm.roll(_currentBlock + 1e6);
175         uint256 startingFee = _pool.getSwapFeePercentage();
176         uint256 amountIn = 1e15;
177 
178         _vault.swap(
179             IVault.SingleSwap({
180                 poolId: _pool.getPoolId(),
181                 kind: IVault.SwapKind.GIVEN_IN,
182                 assetIn: IAsset(address(_token0)),
183                 assetOut: IAsset(address(_token1)),
184                 amount: amountIn,
185                 userData: new bytes(0)
186             }),
187             IVault.FundManagement({
188                 sender: address(this),
189                 recipient: payable(address(this)),
190                 fromInternalBalance: false,
191                 toInternalBalance: false
192             }),
193             0,
194             1e8
195         );
196 
197         uint256 feeAfterSwap = _pool.getSwapFeePercentage();
198         uint256 amountInAfterSwapFee = amountIn.sub(amountIn.mulUp(
                startingFee));
199         assertLt(amountInAfterSwapFee, amountIn);
200         uint256 tradeSize = amountInAfterSwapFee.divUp(1e21);
201         assertLt(startingFee, feeAfterSwap);
202         assertEq(feeAfterSwap, startingFee.add(_pool.
                feeIncreasePerTradeSize().mulUp(tradeSize)));
203     }
204 
205     function testSwapFeeNotGoesAboveMaximium() public {
206         uint256 startingFee = _pool.getSwapFeePercentage();
207         uint256 amountIn = 1e20;
208 
209         _vault.swap(
210             IVault.SingleSwap({
211                 poolId: _pool.getPoolId(),
212                 kind: IVault.SwapKind.GIVEN_IN,
213                 assetIn: IAsset(address(_token0)),
```

```
214                 assetOut: IAsset(address(_token1)),
215                 amount: amountIn,
216                 userData: new bytes(0)
217             }),
218             IVault.FundManagement({
219                 sender: address(this),
220                 recipient: payable(address(this)),
221                 fromInternalBalance: false,
222                 toInternalBalance: false
223             }),
224             0,
225             1e8
226         );
227 
228         uint256 feeAfterSwap = _pool.getSwapFeePercentage();
229         assertEq(feeAfterSwap, _pool.maximumSwapFeePercentage());
230     }
231 }
```

Listing B.1 – Pool tests