CLUE

WHITEPAPER

v. 1.0.0 (Initial Protocol Specification)

https://clue.markets/

@clue_markets

discord.gg/rXKz7maTZD

t.me/clue_markets

reddit.com/u/CLUE_MARKETS

Executive Summary

CLUE — a decentralized protocol for prediction markets without unilateral operator control and with a zero-trust architecture that eliminates the need to trust operators and infrastructure. The protocol is governed through the DAO, the economy is built on transparent fees and the *burn* mechanism, and security is ensured by on-chain moderation and arbitration with stake-backed accountability of participants.

The protocol is created as a universal market infrastructure for on-chain forecasting, in which traders receive transparent and predictable trading conditions, analysts and researchers receive verifiable market signals, market creators receive an open mechanism for creating and launching markets on real-world events, and moderators and arbitrators receive a formalized accountability and rewards system. This model combines trading, expertise and governance in one system, allowing the ecosystem to grow driven by real user activity and the quality of decisions made.

Table of Contents

1. CLUE Overview

CLUE is a decentralized prediction markets protocol in which users trade outcome shares, with market prices implying probabilities of real-event outcomes in politics, sports, economics, crypto, technology and other areas. The protocol is initially designed as an open market infrastructure without intermediaries and without manual intervention or censorship, so key user actions are based on transparent on-chain rules, and not on the decisions of a closed/centralized operations team.

CLUE is based on an architecture without unilateral admin control and with a separation of roles, where the DAO is responsible for the governance parameters and economics of the protocol, and moderators and arbitrators ensure the quality of markets and dispute resolution with stake-backed accountability. The rules of operation, fee logic and resolution procedures are encoded in smart contracts and can be publicly verified, which reduces the risk of arbitrary changes and increases the predictability of the system for all participants.

CLUE creates a unified environment where traders have a clear and verifiable market mechanism, market creators can create markets permissionlessly, and the community is directly involved in moderation, arbitration, and protocol development. As a result, the ecosystem grows not through centralized administration, but through user activity, the quality of the markets created and coordinated economic incentives within the DAO model.

1.1. Core Values

CLUE is more than a prediction markets protocol. It is an ecosystem in which community participation directly influences the development of the platform, and user actions shape its long-term sustainability. There are no centralized administrators or opaque procedures here, because key rules are encoded in smart contracts and verified on-chain.

We are building a space where fairness is ensured by code, participant responsibility is confirmed by economic mechanisms, and incentives are aligned in the interests of the ecosystem as a whole. The value of a token in CLUE is related to actual protocol activity and trading volume, rather than manual metric manipulation. CLUE is a shift from closed solutions to an open market model where the community participates in creating, managing and sharing value.

1.1.1. Mission

Our mission is to create a prediction market in which the rules are transparent, formalized and do not depend on the will of individuals. The economic value of the CLUE token is formed through the actual use of the protocol: trading fees, an on-chain burn mechanism (part of the fees are burned by a special smart contract) and incentives for staking and role-based participation.

1.1.2. Vision

We see CLUE as **a next-generation prediction market platform**, which develops without external manual control. The ecosystem is self-sufficient: it is governed by the DAO and supported by the actions of the participants themselves, including the creation of markets, moderation, arbitration and participation in governance.

- **Governance:** key protocol parameters are changed through the DAO according to a formalized procedure, rather than by centralized decisions of the team.
- **Economy:** Market creators and participants who attract users receive a fair share of fees in accordance with on-chain distribution rules.
- **Deflationary model:** a portion of the protocol's revenue is automatically allocated to the Burn contract, which consistently reduces the supply of CLUE.
- **Autonomy:** The development of the ecosystem is ensured by user activity through the launch of markets, moderation, arbitration and DAO initiatives.
- **Decentralization:** the protocol does not have a single centralized control point, and changes to parameters are possible only through the DAO mechanics.
- **Transparency:** fees, payouts, role actions and results of key procedures are recorded on-chain, and smart contracts are available for public audit.
- **Fairness of incentives:** the economic model aligns the interests of participants around the growth of trading volume and quality of markets, distributing fees between active roles and directing part of it to on-chain burn.

1.2. Technical Independence

CLUE is created as **a protocol without a central operator**, which remains operational even in the complete absence of a central team. Unlike traditional platforms, the system does not rely on a single centralized server or closed APIs, because key business logic is encoded in smart contracts. The backend layer is used only for speed and ease of access: it consists of open nodes and indexers that any participant can run independently and synchronize with the network.

Contracts and protocol parameters can only be changed through **DAO-governed upgrades**. This means that the team does not have the technical ability to manually change fees, resolution logic, arbitration rules or other critical parameters. All changes go through on-chain voting and are executed only within the framework of the approved DAO procedure.

This architecture provides three key benefits.

- **Censorship resistance:** markets cannot be arbitrarily deleted or suspended because their logic and state are enshrined in the blockchain.
- **Transparency and Verifiability:** smart contracts are open to audit, and every change in parameters is recorded on-chain and requires a collective decision by the DAO.
- **Continuity of operations:** the protocol continues to function even if individual nodes or services are unavailable, since there is no single point of failure.

Technical independence makes CLUE truly **a self-sustaining ecosystem**, which develops through user actions - creating markets, trading, moderation, arbitration and Governance - and does not depend on centralized servers or the administrative powers of the team.

1.3. Problem and Solution

The prediction market sector is developing at a rapid pace and at the same time remains relatively young and underutilized. Existing projects suffer from centralization, opaque market and resolution rules, and centralized decisions. CLUE is designed to remove these limitations and offer a truly decentralized, resilient and self-sustaining protocol.

We are not just solving old problems - we are creating a new model: without a mandatory centralized backend, without unilateral team admin keys, with an economy that stimulates growth and reduces risks for the token. CLUE is an ecosystem where every participant can find their role and benefit from transparent rules.

1.3.1. Problem Statement

- **Centralization:** Most projects retain de facto control over markets, allowing markets to be deleted, rules changed manually, and users blocked, even if the underlying logic is formally stated to be decentralized.
- **Opaque resolution:** Individual addresses or groups in a DAO can have outsized influence, influencing market outcomes and undermining trust in the system, especially in controversial or highly liquid events.
- **Weak moderation of market quality:** Without formalized risk labels, users find out late about markets with signs of scam risk, low-quality wording or vague rules, which is why conflicts and the number of disputes after trading increase.
- **Backend dependency:** blocking the frontend or API effectively paralyzes the platform, even if smart contracts continue to work on-chain.
- **Marketing and traffic:** projects often fail to attract a sufficient number of users and markets, so liquidity remains low and spreads in the long-tail segment become too wide for effective trading.
- **Token depreciation:** Many protocols distribute native tokens as the main reward, after which users trigger mass sell-offs, creating sustained pressure on the price and a disconnect between product activity and token value.
- **Irrelevance of content:** on many competing platforms, market topics are curated by the core team, which leads to a delayed launch of trending events and a decrease in user engagement, and liquidity is ultimately concentrated in only a few markets.
- **Dependence on external liquidity:** traditional prediction markets often need large market makers and external support, without which a significant part of the markets quickly loses depth and activity.

1.3.2. Solution

- **Separation of roles of DAO and operational participants:** The DAO in CLUE is responsible for governing protocol parameters and fee economics, but does not replace moderators and arbitrators, which eliminates role conflict and reduces the risk of concentration of influence in one subsystem.
- **On-chain moderation with risk labeling:** staked moderators label markets as scam-risk or low-quality, and participants receive warnings before entering a position, which reduces the number of weak launches and improves the quality of content at an early stage.
- **Arbitration with economic accountability:** if there is disagreement on the outcome, a dispute is initiated, and once the threshold is reached, arbitration is escalated to an arbitration committee (default size: 5, DAO-configurable); outcomes are selected by weighted voting, with finalization gated by the consensus threshold, and erroneous actions by moderators and arbitrators are penalized via slashing.
- **AI assistant support in the reference interface and supporting infrastructure:** The AI assistant analyzes the completeness of rules and the quality of market formulations, offers recommendations on risk classification and enhances user awareness without replacing final on-chain procedures.
- **Open access:** any participant can create a market, participate in moderation or arbitration if staking conditions are met, and also make a proposal to the DAO.
- **Verifiability:** all transactions, roles and rules are recorded in the blockchain, and the data is available for public audit and independent verification.
- **Audience growth:** the built-in referral system rewards referrers and active users for attracting new traders and market creators, and in the absence of a referral, the corresponding share of the fee is sent to the DAO treasury.
- **Token protection through utility demand:** CLUE is used for staking, voting, role-based participation, trading mechanics and discounts, which creates a stable demand for the token within the protocol and reduces dependence on short-term incentive campaigns.
- **Community content:** the protocol does not restrict topic selection to a centralized editorial team, because market topics are formed by users, and market creators are rewarded for popular and high-quality markets.
- **LMSR-based AMM:** protocol uses LMSR with dynamic parameter *b*, which supports more stable pricing and continuous liquidity, including in long-tail markets where classic models often require external market making.
- **Infrastructure sustainability:** critical data and protocol logic are hosted in smart contracts and IPFS, and the backend is used as a fast indexing layer rather than a mandatory trust point.

As a result, CLUE becomes **a sustainable, self-sufficient protocol**, where trust in resolution, quality of markets, liquidity and token value are built into a single on-chain economy. The protocol grows with user activity and the quality of decisions made, and not at the expense of external capital or manual control.

1.3.3. CLUE Differentiators

CLUE is not just another prediction markets protocol. It is an ecosystem in which each element is built into a single self-sustaining model. The key advantage of CLUE is not only in the technology, but also in the way the protocol builds the interaction of users, roles and economic incentives on transparent on-chain rules.

Earning rewards from moderation and arbitration

Instead of hidden administrators and opaque decisions, moderation and arbitration are carried out by community members. A user who stakes CLUE and meets the role requirements can participate in market moderation and dispute resolution. This approach turns moderation and arbitration into a full-fledged economic loop: honest/accurate decisions are rewarded, and erroneous and dishonest decisions are punished by slashing mechanics.

DAO as governance layer

CLUE is managed by a DAO, where protocol parameters are changed through formalized on-chain procedures, rather than manually by a core team. This applies to key system settings, including fees, economic parameters and upgrade rules. The DAO in CLUE is responsible for protocol governance and distribution mechanics, but does not replace moderators and arbitrators in operational resolution processes.

Burn contract and deflationary logic

A portion of the fees is automatically sent to a special burn contract, which burns CLUE according to predetermined rules. As a result, the increase in user activity and trading volume is accompanied by a reduction in the circulating supply of the token. The deflationary component is built into the protocol at the smart contract level and does not depend on manual decisions.

AMM and default liquidity

CLUE uses an LMSR-based AMM with dynamic parameter *b*, which maintains continuous liquidity and more stable pricing, including long-tail markets. The protocol reduces dependence on external market makers and creates predictable trading rules within its own on-chain infrastructure.

Referral growth system

CLUE rewards not only trading activity, but also contribution to the expansion of the ecosystem. Participants who refer new users receive a share of fees according to transparent distribution rules. If there is no referral, the corresponding share is sent to the DAO treasury, strengthening the collective economy of the protocol.

Rewards for market creators

In CLUE, market topics are formed not by a centralized editorial team, but by the users themselves. Market creators receive a share of fees from activity in their events, so economic motivation is directly related to the quality of wording, relevance of topics and audience engagement. This makes the content layer of the protocol active and adaptive to the current market agenda.

Decentralization at all levels

CLUE does not rely on a privileged centralized backend, hidden servers or unilateral team admin keys. The protocol's critical logic, rules, and market conditions are encoded in smart contracts, and the data is hosted in a decentralized storage infrastructure. Backend nodes perform the role of indexing and speeding up interfaces, while any participant can run their own node without receiving special privileges in relation to contracts.

EVM-based architecture

CLUE is deployed in an EVM-compatible environment and uses standard ecosystem tools. This ensures compatibility with proven infrastructure, simplifies integrations, and creates a strong foundation for scaling. For users, this means access to prediction markets with low transaction costs and high throughput at the core network level.

As a result, CLUE forms **a high-performance and self-sufficient prediction markets platform**, where transparent rules, cryptoeconomic security and decentralization are combined into a single operational model. The protocol is equally suitable for retail participants and professional teams who value scalability, interoperability and sustainable liquidity.

2. Ecosystem and Product

The **CLUE** ecosystem is not just a prediction markets platform, but a protocol with a unified on-chain logic layer, where each role contributes to shared value creation: market creator, trader, moderator, arbitrator, staker, referral partner and DAO participant. The operating rules are encoded in smart contracts, so the development of the system is determined by the actions of users and transparent economic incentives.

The CLUE architecture shapes **an organic flywheel** along the chain:
more markets → more referrals → higher trading volume → more fees → more CLUE burn → higher rewards → more users → new markets. This cycle combines content, distribution, liquidity and tokenomics into a single growth model that intensifies as the ecosystem becomes active.

Result: the value of CLUE is tied to the actual use of the protocol:
growth of markets and audience → growth of transactions and fees → strengthening of deflationary tokenomics → strengthening of long-term incentives for all roles.

2.1. Key Ecosystem Elements

CLUE relies on several mutually reinforcing pillars. Together they create a self-sustaining system in which participants' incentives are aligned with protocol growth.

- **Markets and content economy.** Any user can launch a market with a clear description and a source of truth—and receive a share of fees from its trading volume. Trends emerge in the community rather than being imposed centrally.
- **Default AMM liquidity.** The automated market maker provides quotes and execution even under moderate liquidity, making entry easier for newcomers and pricing more predictable.
- **Stake-backed moderation.** Decision quality is enforced economically: correct decisions are rewarded, while abuse is penalized via slashing.
- **Referral growth system.** Any user can become an ambassador and earn a share of fees from referred traders' real trading volume. Organic distribution is built into the protocol.
- **Burn mechanism.** A portion of fees is routed to a dedicated smart contract that burns CLUE tokens. Higher usage strengthens deflation.
- **DAO governance.** Protocol parameters, treasury budgets, and partnerships are set by the community through on-chain voting with governance delay protection (grace + executionWindow).
- **EVM compatibility.** The protocol is deployed in an EVM-compatible environment, preserving interoperability and on-chain verifiability.

2.2. User Journey

CLUE is designed so that the actions of each participant create value not only for them, but also for the protocol as a whole. Every new market, every trade and every staking action strengthens retention, reduces circulating supply (free float), and supports the trust and economic sustainability of the ecosystem.

- **Market creator → Monetization of content and launch of new markets.**
  The creator launches markets tied to real-world events, around which trading and liquidity form. The higher the quality and relevance of the markets, the greater the volume of transactions and fee flow.
  Effect: the growth in the number of quality markets expands the content pool, increases trading volume and strengthens the network effect of the protocol.
- **Trader → Trading, hedging and price signals.**
  Traders create liquidity, maintain price dynamics and create the main fee flow. Active trading improves the efficiency of markets and makes forecasts more informative.
  Effect: more trades mean more fees, which strengthens deflationary tokenomics and maintains the value of the token.
- **Moderator → Quality control and early risk signals.**
  Moderators stake CLUE, mark markets by risk level, and help filter out weak or questionable launches. Slashing is used for erroneous or dishonest actions.
  Effect: higher quality of markets, lower share of conflict cases, higher user confidence in the platform.
- **Arbitrator → Dispute resolution and finalization of outcomes.**
  Arbitrators are involved when a dispute escalates and make decisions based on a formalized procedure and stake-backed accountability. Correct decisions are rewarded, wrong decisions are punished by slashing mechanics.
  Effect: Robust and verifiable resolution reduces the risk of abuse and strengthens the legitimacy of outcomes.
- **Discount staker → Reduced trading costs.**
  users lock CLUE to receive fee discounts and more profitable trading. This increases the retention of the token within the ecosystem.
  Effect: trading volume is stimulated, and the share of tokens in staking reduces the liquid supply in the market.
- **DAO participant → Manage protocol parameters.**
  CLUE owners participate in voting on fees, revenue distribution and key system settings. Changes go through transparent on-chain procedures.
  Effect: Predictable governance increases protocol stability and reduces centralized risks.
- **Referral partner → Organic audience growth.**
  Partners attract new users and market creators, expanding the active network of participants. The referral model turns the community into a scaling channel.
  Effect: lower cost of acquisition, higher growth rates and stronger network effect of the platform.

The result is a closed cycle of growth in which market creators expand event offerings, traders create liquidity and fees, moderators and arbitrators maintain quality and trust, stakers strengthen tokenomics, referral partners scale audiences, and DAO ensures transparent rule development. The higher the activity of participants, the stronger the protocol becomes at the level of economics, governance and market sustainability.

2.3. Competitive Positioning

CLUE occupies a special place among prediction markets because it is not only a platform for trading outcome shares on event markets, but a base-layer protocol for creating independent markets and interfaces. Any participant can use the CLUE infrastructure, deploy their own node, operate under their own brand and build a product on top of a common protocol layer. At the same time, the foundation of the ecosystem remains the same: the CLUE native utility token, on-chain rules, DAO governance, moderation and arbitration with stake-backed accountability.

Unlike traditional centralized or semi-decentralized solutions, CLUE is built as a self-contained network of interoperable instances. This means that new interfaces and local operators do not fragment the ecosystem, but expand it, increasing overall trading volume, liquidity and network effect. CLUE competes not only with blockchain prediction platforms, but also with classic betting, offering a model without censorship, hidden rules and a single centralized control point.

Governance and trust

Most competitors retain control over a team or a limited number of validators, so key processes ultimately depend on the platform operator. In CLUE, the basic parameters of the protocol are fixed in smart contracts and are changed only through a DAO with a timelock, and the roles of the DAO, moderators and arbitrators have distinct responsibilities and do not overlap. This makes the rules predictable for all nodes and interfaces built on CLUE, and builds trust in the protocol as a single on-chain standard.

Liquidity and trading

On traditional platforms, liquidity often depends on a single operator or external support from market makers. In CLUE, trading and pricing are built on a common protocol base with the LMSR AMM model and dynamic parameter *b*, so the network maintains continuity of trading even with different activity in individual interfaces. Since different products and nodes operate on the same economic foundation, the growth of each instance enhances the overall market depth and liquidity stability across the entire ecosystem.

Token Economics

Many projects drive activity with massive airdrops, which often creates sell pressure and a disconnect between product metrics and token value. In CLUE, the token is built into the utility loop of the protocol: it is used in trading, role staking, governance and discount mechanics. Fees are distributed according to transparent on-chain logic, and part of the trading fees is sent to the burn mechanism, so the value of the token is related to the real activity of the network, and not to short-term incentive cycles.

Attracting an Audience

CLUE does not rely on centralized marketing budgets, but on a decentralized growth model through interface operators, referral partners and market creators. Participants who refer users and create demand can monetize this contribution through fees, including shares associated with the referral model and creator fees. This approach turns the community and local products into a scaling engine, where the growth of the audience in individual nodes is converted into the overall growth of the protocol.

Operators, nodes and legal scaling

The CLUE protocol architecture paves the way for the launch of independent products in different regions and jurisdictions. Operators can build their own interfaces and operational processes, adapting them to local requirements and, if necessary, obtaining regulatory approvals at the jurisdiction level. Structurally, this is close to a white-label operator model, where local distribution and compliance are developed on the operator’s side, and the technological and economic base remains common and compatible within CLUE.

As a result, CLUE becomes **a self-sufficient network protocol**, where governance, liquidity, token economics and audience expansion are combined into a single growth cycle. The ecosystem scales simultaneously along the product, economic and legal axes, while maintaining uniform on-chain rules and compatibility between all instances.

3. Economic Model

CLUE is a protocol where each transaction mechanically increases the scarcity of the token through on-chain burn: part of the fee automatically goes into a burn contract that burns CLUE tokens, and each participant who creates real value receives direct revenue:

- **Market creation:** 500 CLUE - immediately and entirely to the moderators.
- **Basic trading fee per trade:** 3%

  The rate is fixed on-chain and changes only through the DAO.

  Fee distribution:

  - **Burn treasury:** 50% fee → 1.50% of volume is burned.
  - **To the market creator:** ≈21.67% fee → ≈0.65% of volume
  - **Arbitration (upon appeal):** if the appeal is upheld, the share of the creator (≈0.65% fee) is forwarded to the arbitrators in proportion to their weights, or the committee receives the appellants' appeal bonds (10% stake, minimum 100 CLUE, maximum 500 CLUE) for a specific arbitration case.
  - **Moderators:** ≈3.33% fee → ≈0.10% of volume.
  - **Referrals:** 25% fee → 0.75% of volume — paid if a referral partner is specified.
  - **DAO Rewards (if there is no referral):** 25% fee → 0.75% of volume.

  DAO rewards are distributed among DAO participants in proportion to their weight.

3.1. Economic Principles and Philosophy

- **Reward for value created.** The CLUE economy directs income to those participants who develop the ecosystem: create high-quality markets, ensure fair procedures and attract new audiences.
- **Rising trading volume increases scarcity.** Part of the fees is sent to the burn mechanism. As trading activity increases, the volume of tokens in circulation decreases.
- **On-chain transparency and verifiability.** Key economic flows and parameters are fixed in smart contracts, and changes are made only through the DAO procedures.
- **Growth flywheel.** Creators publish more quality markets, referral partners bring new audiences, which increase trading activity and overall transaction volume; higher volume increases fee flow and CLUE burn volume; decreased circulating supply increases rewards for active roles; stronger incentives attract even more users, after which the cycle repeats at a new level.

3.2. Roles and Incentives: Who Benefits and Why

- **Market creator** gets a share 0.65% from the trading volume of their markets and monetizes the quality and relevance of the content. If the dispute escalates and the appeal is upheld, the share may be redirected to the arbitration process.
- **Moderators** receive fee payouts and rewards for processing market listings. The basic share of trading volume is ≈0.10%, and the rewards for processing market listings are formed from a fixed fee 500 CLUE, which is distributed among moderators according to weights. Slashing is used for dishonest or systematically erroneous decisions.
- **Referral partner** receives 0.75% from the trading volume of invited traders. If there is no referral, their share is automatically sent to DAO Rewards.
- **Trader** trades through AMM with continuous liquidity, pays a transparent fee 3% and can reduce costs through a system of staking-based discounts.
- **Arbitrator** participates during dispute escalation and receives a reward for correct analysis of the case from the arbitration pool, including the redistributed share of the creator and appellant bonds according to the rules of the protocol.
- **DAO** receives 0.75% fees in transactions without a referral, manages protocol parameters and the treasury budget, including 40% of total supply, and is interested in sustainable growth of trading volume and quality of markets.

3.3. Market Creation Fee as an Economic Access Layer

Publishing each market costs P = 500 CLUE. This amount is 100% distributed among moderators who accepted the market for processing in proportion to their active weights.

Notation

- **K** - number of moderators
- **w_k** — active weight of moderator k
- **$W = \Sigma w_k$** - sum of weights

Market creation fee distribution formula

`Payout_pub(k) = 500 × (w_k / W)`

Economic effects

- **Moderation payouts** create a sustainable on-chain reward model without manual subsidies.
- **Market quality filtering** reduces the share of weak and spam markets.
- **Scalability** increases payments to moderators in proportion to the increase in the number of market listings.

Current restrictions: minimum market duration - 3 days, minimum interval between market close and event occurrence - 3 hours, maximum outcomes in one market - 15.

Trading fee

Each trade volume V is subject to a base fee 3%. Effective fee taking into account discounts: `f_eff = max(0, 3% - discount)`, where `discount` is a personal discount from Fee Discount slots.

Actual fee paid

`feeTokens = V × f_eff`

All shares below are calculated from `feeTokens`. Any rounding residuals are sent to the burn allocation.

Fee split

- **Burn Treasury** — **50%** fee, which corresponds to 1.50% of V at a fee rate of 3%
- **To the creator of the market** — **≈21.67%** fee, which corresponds to ≈0.65% of V at a fee rate of 3%
- **Moderators** — **≈3.33%** fee, which corresponds to ≈0.10% of V at a fee rate of 3%
- **Referral** — **25%** fee, which corresponds to 0.75% of V if you have a referral
- **DAO Rewards** — **25%** fees in transactions without a referral, which corresponds to 0.75% of V

Example

- **Transaction volume V:** 100 000 CLUE
- **Effective rate f_eff:** 3%
- **Fee feeTokens:** 3 000 CLUE

Distribution by example

- **To the market creator:** `3 000 × 21.67%` ≈ 650 CLUE
- **Moderators:** `3 000 × 3.33%` ≈ 100 CLUE
- **Referral partner:** `3 000 × 25%` = 750 CLUE (if there is a referral)
- **DAO Rewards:** `3 000 × 25%` = 750 CLUE (if there is no referral)
- **Burn Treasury:** `3 000 × 50%` = 1 500 CLUE

If there is no referral, their share of the fee is sent to DAO Rewards. The Burn share remains unchanged and is used to burn CLUE according to the rules of the protocol.

3.4. Staking-Based Discounts (Fee Discount Slots)

The discount system is based on stake ranking. There are 10 slots available and a total of 25 seats. Slot parameters and distribution rules can be changed through DAO voting.

# slots	Discount	Capacity
1	99%	1st place
2	90%	1st place
3	80%	1st place
4	70%	2 places
5	60%	2 places
6	50%	2 places
7	40%	3 places
8	30%	3 places
9	20%	5 seats
10	10%	5 seats

- Discounts are fixed per slot: `99%, 90%, 80%, 70%, 60%, 50%, 40%, 30%, 20%, 10%`
- A participant can enter a slot by displacing the stake of the participant with the minimum stake in the corresponding slot
- The minimum stake lock period is 30 days, and the unstaking cooldown is 7 days. These parameters can be adjusted by DAO.

All slot and discount parameters are determined and updated through the DAO voting.

4. Tokenomics

CLUE is a utility token for governance and incentives: voting in DAO, trading fee discounts through staking, access to moderator/arbitrator roles. Vesting aligns interests over the long term. A burn contract maintains value as activity increases.

- **Token name:** CLUE
- **Ticker:** CLUE
- **Total supply:** 500 000 000
- **Initial supply:** 500 000 000 CLUE is completely minted to the deployer and subsequently distributed into pools according to tokenomics; the max supply is equal to the initial supply, 18 decimals, the burn and pause functions are available to the owner.
- **Distribution mechanism:** Liquidity & MM Private Round Public Distribution Core Team DAO Treasury Operational Reserve

4.1. Token Utility

CLUE is the protocol's operational token. Its value is built not on abstract promises, but on the specific actions of users within the system. The token is required for trading, accessing role-based features, receiving fee discounts, and participating in governance through the DAO. The higher the activity in the protocol, the higher the practical utility of the token.

- **Trading and settlements within the protocol**
  CLUE is used in staking and trading on the protocol's markets. The user uses the token as the primary asset when opening and closing positions, as well as when redeeming payouts after market resolution. This makes the demand for CLUE functional because it is related to actual trading activity and not just speculation.
- **Access to moderator and arbitrator roles**
  To become a moderator or arbitrator, a participant must stake CLUE. Stake serves as economic accountability: correct actions support role reputation and rewards, while erroneous or dishonest decisions lead to slashing. As a result, the token acts not only as a medium of exchange but also as a trust and integrity mechanism.
- **Reducing trading costs through staking**
  Active users can lock/stake CLUE and receive discounts on trading fees based on the tier model. The higher the staking level, the lower the final fee on transactions. This increases trading efficiency for active participants while creating an incentive to hold the token in the protocol.
- **DAO participation and protocol governance**
  CLUE owners participate in governance processes and vote on key parameters of the system. Through the DAO, decisions are made on fees, economic settings, treasury directions, and protocol logic updates. Thus, token holders participate in the development of CLUE not formally, but through real on-chain governance mechanisms.

As a result, CLUE's utility is based on four clear functions: trade, roles, discounts and governance. This creates a sustainable model in which the token is constantly used in the operational life of the protocol and is directly linked to its growth.

4.2. Deflationary Model

The CLUE economy is built on the principle of programmatic scarcity. This means that as user activity increases, the number of tokens in free circulation decreases, strengthening scarcity dynamics.

The main deflation mechanism is on-chain burn through a special smart contract. Part of the protocol fees are automatically sent to a burn contract, which burns CLUE tokens according to specified rules. As a result, every trade, every new market and every attracted trader reduces the supply of CLUE, increasing its scarcity.

The second source of scarcity is staking roles and discounts. Tokens locked by moderators, arbitrators and traders are effectively excluded from free circulation, creating the effect of a temporary lock-up of liquidity. As the ecosystem scales, this results in a steady decline in available supply.

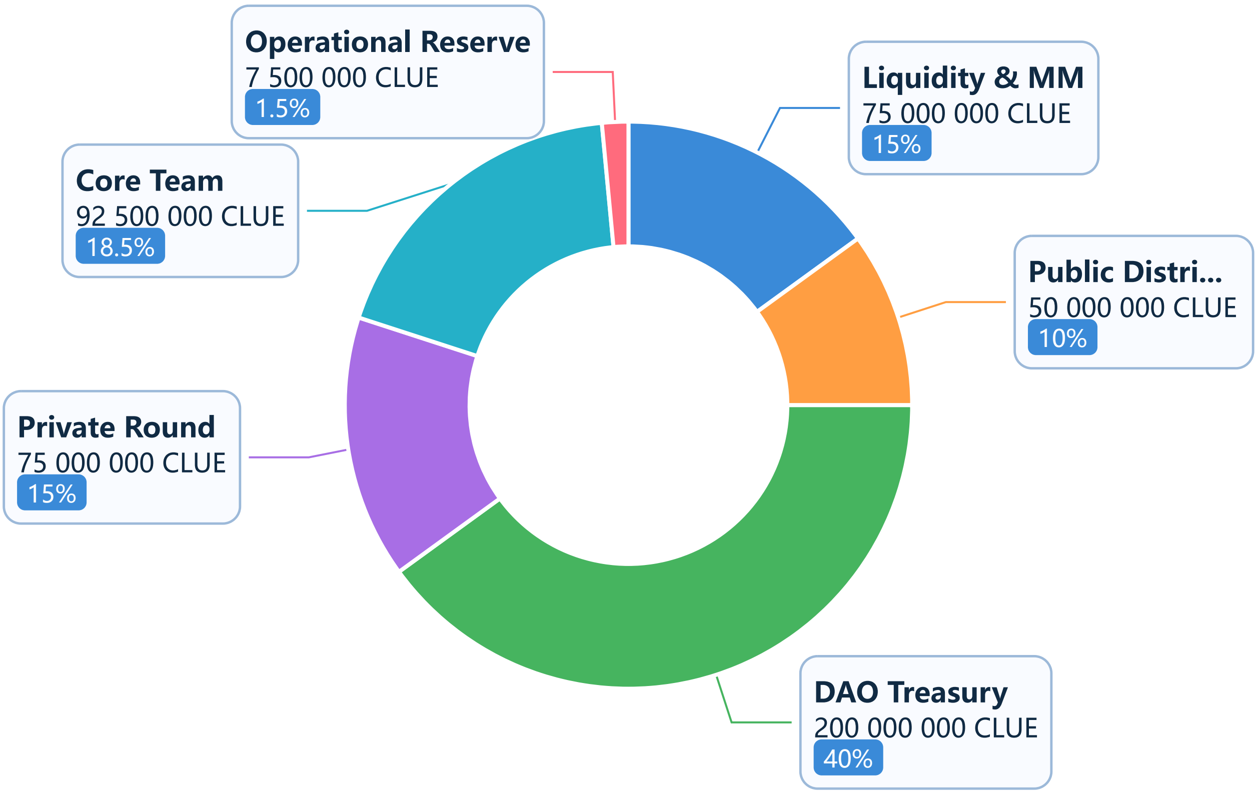
The third element is the absence of inflationary token emissions. Unlike projects that give away tokens as rewards (and thus cause price pressure), CLUE builds an economy based on sustainable fee flows and the distribution of real protocol revenue. This makes the model predictable and reduces sell-pressure incentives.

Together, these mechanisms turn CLUE into a deflationary asset, the value of which is directly related to the growth of trading volume and involvement. The more actively the protocol is used, the smaller the supply of the token becomes, and the higher its value for all participants in the ecosystem.

4.3. Token Distribution by Pools

Total token supply **CLUE** is **500 000 000** CLUE tokens. Each pool plays a clearly defined role in the ecosystem and balances the interests of traders, moderators, investors, and the DAO. The distribution is designed to simultaneously stimulate growth at the start, ensure long-term sustainability and eliminate the risks of concentration of power in the hands of the team or large investors.

On-chain note: pool amounts in this section are allocation targets. In current contracts, these values are initialized/funded by transactions and are not hardcoded immutable constants per pool.



Category	%	Tokens	
Liquidity & MM	15.00%	75 000 000	*Geometric decay unlock*
Private Round	15.00%	75 000 000	*12-month cliff + 24-months linear unlock**
Public Distribution	10.00%	50 000 000	*To Be Determined* (depending on the exchange/platform and listing)*
Core Team	18.50%	92 500 000	*12 months + 48-months linear unlock*
DAO Treasury	40.00%	200 000 000	*Geometric decay unlock*
Operational Reserve	1.50%	7 500 000	*Flexible allocation*

4.3.1. Liquidity & MM

Volume: 75 000 000 CLUE

Contract: `LiquidityPool`

Unlock: 0.014%/day (≈5%/year, geometric unlock every 24h from the remaining balance)

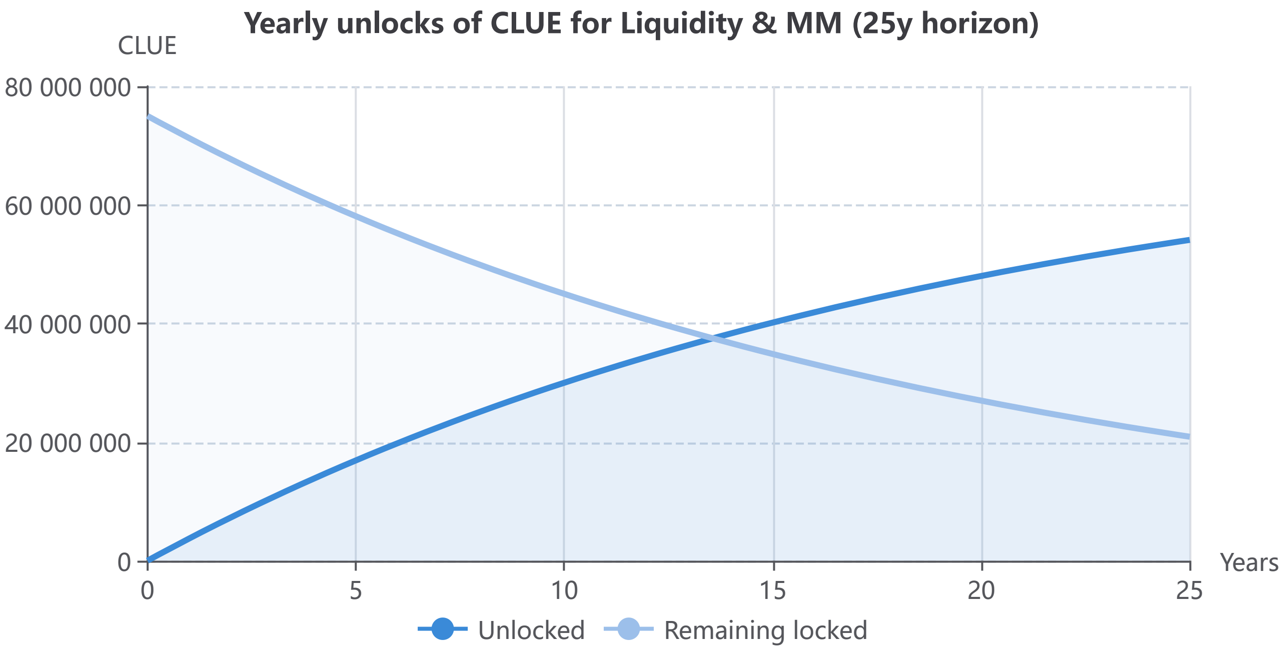
The **Liquidity & MM** pool is allocated for initial and long-term liquidity. The contract releases tokens on an unlock schedule without cliff unlocks or external trigger dependencies.

Purpose

- **Market depth support:** creating sustainable liquidity in CLUE trading pairs on DEX and, if necessary, on CEX.
- **Stabilization of trading conditions:** reducing spreads and slippage to provide better order execution for traders.
- **Market making operations:** ensuring continuous quotes and operational market infrastructure during periods of low and high activity.
- **Smooth token release:** daily geometric unlock limits sudden changes in circulating supply and increases the predictability of token flow.
- **Transparency of use:** the movement of pool funds and liquidity transactions are recorded on-chain and are available for public verification.

Control and spending

The `withdraw()` call is only available to the treasury owner **(DAO)** and within `availableToWithdraw`; attempts over the limit are rejected (`AmountExceedsUnlocked`). Each movement is logged by the event `LiquidityWithdrawal`; spending allocations are approved by vote.



4.3.2. DAO Treasury

Volume: 200 000 000 CLUE

Contract: `DAOTreasury`

Unlock: 0.008%/day (≈3%/year, geometric unlock every 24h from the remaining balance)

DAO Treasury is a strategic reserve of the protocol, which is spent only on on-chain DAO decisions with governance delays (grace + executionWindow). The pool is not used as a source of arbitrary distribution and is not intended for manual price intervention. Its mission is to finance the long-term sustainability of CLUE at the level of infrastructure, security and ecosystem growth.

Purpose

- **Protocol development:** financing smart contract upgrades, new modules and technical improvements approved by the DAO.
- **Security and reliability:** audit, bug bounty, risk monitoring and other measures that increase the stability of the protocol.
- **Reserve for critical initiatives:** Funding high-impact priorities that directly impact protocol sustainability, user safety, and the continuity of the CLUE ecosystem.
- **Ecosystem Grants:** supporting teams and contributors who create useful integrations, analytics tools, interfaces and infrastructure services for CLUE.
- **Regulatory and legal support:** funding legal counsel for the protocol, preparation of legal opinions, analysis of regulatory requirements in target jurisdictions and operational actions necessary for the legal and sustainable development of the CLUE ecosystem.
- **Partnership programs:** financing initiatives that expand the distribution of the protocol and increase real on-chain trading volume.
- **DAO operating budget:** covering expenses necessary for the implementation of governance decisions and the functioning of DAO processes.
- **Strategic stability reserve:** limited application by DAO decisions for protocol protective and stabilization measures in exceptional scenarios.

Unlocking (geometric decay unlock schedule)

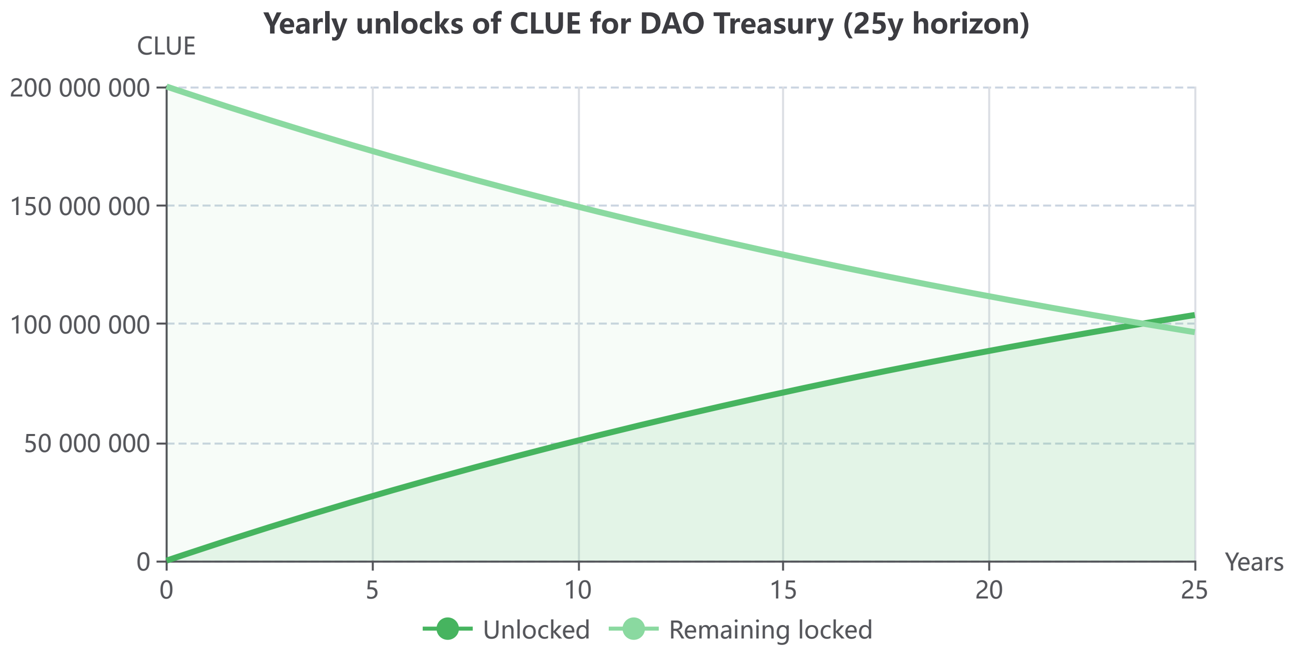
Every 24 hours 0.008% of the remaining balance is automatically unlocked.

- **Total:** $200{,}000{,}000$ CLUE.
- **releaseRate:** `RELEASE_RATE_BPS = 8` with `BPS_DENOM = 100{,}000`.
- **Daily Unlock:** $\mathrm{Release}_d = \mathrm{Remaining}_{d-1} \times 0.00008$.
- **Cumulatively for *d* days:** $\mathrm{Unlocked}(d) = \mathrm{Total} \times \left(1 - (1 - 0.00008)^d\right)$.
- **Available for withdrawal:** $\mathrm{Available} = \max(0,\ \mathrm{Unlocked} - \mathrm{Withdrawn})$.
- **Normalization by day:** periods are counted from the treasury contract `startTimestamp` (deployment day rounded to 00:00) and calculated in full 24h intervals. No explicit period cap is set in the contract.

Effect: in the first year (~365 days) unlocked ≈ 2.9% pool (~ 5.8 million CLUE at 200 million), the rest goes into circulation gradually along a geometric curve.

Control and spending

The `withdraw()` call is only available to the treasury owner **(DAO)** and only within `availableToWithdraw`. Attempts to withdraw an amount over the limit are rejected (`AmountExceedsUnlocked`). Each movement is recorded by the `TreasuryWithdrawal` event, and spending allocations are approved by governance voting and executed through governance delays (grace + executionWindow).



4.3.3. Private Round

Volume: 75 000 000 CLUE

Contract: `PrivateSale` (presale module, separate package)

Unlock: 12m cliff + 24m linear

In CLUE, **Private Round** is structured as a mechanism for strategically forming a core of long-term participants in the protocol. Its task is not only to attract capital, but also to select partners who are ready to participate in the governance process, staking and development of the ecosystem over a multi-year horizon. Participants purchase NFT allocations that secure the right to phased access to CLUE after staking is activated. Funds from the purchase are immediately sent to `treasury`, and CLUE tokens from the Private Round are not intended for direct market sale and are used through targeted staking mechanics.

Purpose of Private Round

Private Round is designed to build a sustainable base of strategic DAO participants who are interested in quality of governance, long-term value of the protocol and responsible decision-making. Thus, the round does not create short-term speculative demand, but a governance and economic loop in which partners are synchronized with the growth of CLUE and participate in the development of the protocol through on-chain processes.

Purpose of funds

- **Security:** external audits of smart contracts, repeated checks before key releases and bug bounty programs.
- **Product development:** financing the development of protocol modules, indexing infrastructure, analytical services and user interfaces.
- **Go-to-market:** marketing, community initiatives, partnership programs and distribution to attract traders, creators and infrastructure participants.
- **Listing and integrations:** technical and operational preparation for CEX/DEX platforms, as well as ecosystem integrations that expand access to CLUE.
- **Operational stability:** ensuring key team and organizational processes necessary for the execution of the roadmap and the continuous development of the protocol.

Vesting and formulas

The countdown of cliff and subsequent vesting begins from the moment staking is activated (`activateStaking`). Tokens are not unlocked for the first 12 months, then a uniform linear unlock is applied over 24 months.

- **Total:** $75{,}000{,}000$ CLUE.
- **cliffDays:** $\boxed{365}$; **linearDays:** $\boxed{730}$.
- **Unlocked:**

$$\text{Unlocked}(d) = \begin{cases} 0, & d < \text{cliffDays} \\ \text{Total} \times \min\left(1, \dfrac{d - \text{cliffDays}}{\text{linearDays}}\right), & d \geq \text{cliffDays} \end{cases}$$

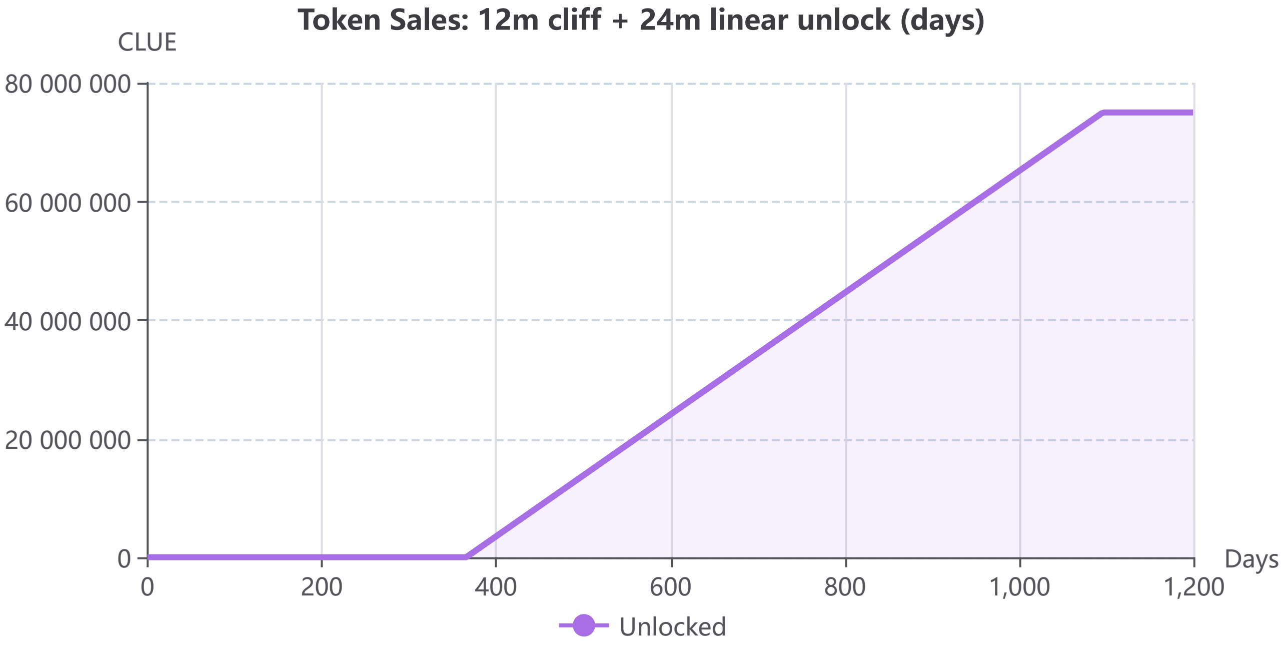
- **Available for staking:** $\text{Available}(d) = \max\big(0,\ \text{Unlocked}(d) - \text{Allocated} - \text{Staked}\big)$. There is no provision for release into free market circulation; tokens are sent to permitted staking pools.

Contracts and restrictions (`PrivateSale`)

- `purchase(tier)` issues an NFT allocation at a fixed price of the selected tier, and the received funds are transferred to `treasury`.
- `activateStaking()` sets the vesting start date and transfers the volume `75,000,000 CLUE` into the contract for phased staking.
- `redeem(tokenId)` accepts the NFT back, burns it and credits the user with a staking balance according to the available volume.
- `stakeDao` / `stakeFeeDiscount` send the available share to DAO/Fee Discount taking into account cliff/vesting and parameters `minWithdrawTimestamp` and `vestingPeriod`.
- Sales can be paused, and tier prices only allow upward movement.

Token flow

Purchasing NFT through `purchase` → activating staking through `activateStaking` → redeeming NFT through `redeem` and accruing staking balance → sending tokens to DAO / Fee Discount according to the rules of target pools (vesting, unbond, slashing, discount slots). This architecture aligns the interests of Private Round participants with the long-term governance of the protocol and the sustainable growth of the entire CLUE ecosystem.



4.3.4. Public Distribution

Volume: 50 000 000 CLUE

Unlock model: TBD (finalized before TGE, taking into account the venue/exchange requirements and listing parameters)

Public Distribution is a public distribution stage of CLUE, designed to provide early participants with wide access to the token and launch token circulation and market trading. The purpose of this pool is to create transparent price discovery, expand the holder base and prepare liquidity for a stable start to trading after the TGE.

Purpose

- **Wide distribution:** provide access to CLUE for retail and professional participants on public and transparent terms.
- **Price discovery:** establish a market price based on open supply and demand without closed allocations outside the stated rules of the round.
- **Initial liquidity provisioning:** provide liquidity for launching trading pairs and support a stable start of trading activity on trading platforms.
- **Ecosystem growth:** expand the active audience of the protocol, including future traders, DAO participants, moderators, arbitrators and referral partners.

Principles of the final unlock model

The exact Cliff and Vesting parameters are finalized shortly before listing, but are approved within the framework of the following principles:

- **Market stability:** the model should not create excessive short-term pressure on the price in the early stages of trading.
- **Compatibility with listing venue requirements:** the final distribution model is formed taking into account the rules of the selected exchange/platform.
- **Synchronization with liquidity:** The unlock schedule is consistent with the plan for the phased launch of trading and the actual depth of the market on the selected platforms, so that the supply of tokens enters circulation in a balanced manner and does not degrade liquidity quality in the first stages of listing.
- **Community Transparency:** The final terms of Cliff/Vesting are published before the TGE and are recorded in official documentation to ensure that the rules are not unexpectedly changed after publication.

Basic scenarios considered for launch

- **Scenario A:** partial unlock at TGE followed by linear vesting.
- **Scenario B:** short cliff with phased unlock to reduce early volatility.
- **Scenario C:** full unlock at TGE, if the platform format requires it and the starting liquidity is sufficient for stable trading.

The final configuration is selected based on the long-term stability of the protocol and the quality of the market launch. Public Distribution in CLUE is not seen as a one-time sale, but as an entry point for new audiences into the economy, governance and operational roles of the ecosystem.

4.3.5. Team Allocations

Volume: 92 500 000 CLUE

Contract: `DAOPool`

Unlock: 12m cliff + 48m linear (from TGE)

The team allocation is deposited in `DAOPool` as a long-term staking allocation with a schedule starting from TGE. Cliff vesting is applied for the first 12 months, after which linear vesting is applied for 48 months. This structure secures the team's long-term commitment to the protocol and reduces the risk of early market pressure.

Contract note: `DAOPool` supports generic per-stake `minWithdrawTimestamp` + `vestingPeriod`. The 12m/48m profile is a governance allocation schedule, not a hardcoded constant inside `DAOPool`.

Purpose

Team Allocations enshrines the CLUE principle, in which the team is part of the protocol community and participates in DAO governance according to the same transparent on-chain rules as governance partners. This allocation creates a long-term model of responsibility, where the team is interested in the quality of decisions made through voting and in the sustainable growth of the value of the ecosystem. The priority remains the development of the protocol, strengthening governance layers and consistent work in the interests of the community.

- **Participation in DAO based on uniform rules:** the team participates in governance processes as a long-term holder of CLUE and acts within the same on-chain procedures as other participants.
- **Long-term retention incentive:** The cliff and vesting parameters in `DAOPool` maintain a focus on protocol development and eliminate the incentive to sell tokens early.
- **Aligning interests with the community:** The team's financial outcome is directly related to the growth of activity, user trust and the sustainability of the CLUE tokenomics.
- **Roadmap execution support:** The allocation ensures operational continuity for protocol development, security, and scaling.

Schedule parameters

- **Total:** $92{,}500{,}000$ CLUE.
- **cliffDays:** 365.
- **linearDays:** 1 460 (48 months after cliff).
- **Schedule start:** from the TGE date.

Formulas and access

- **Unlocked:**

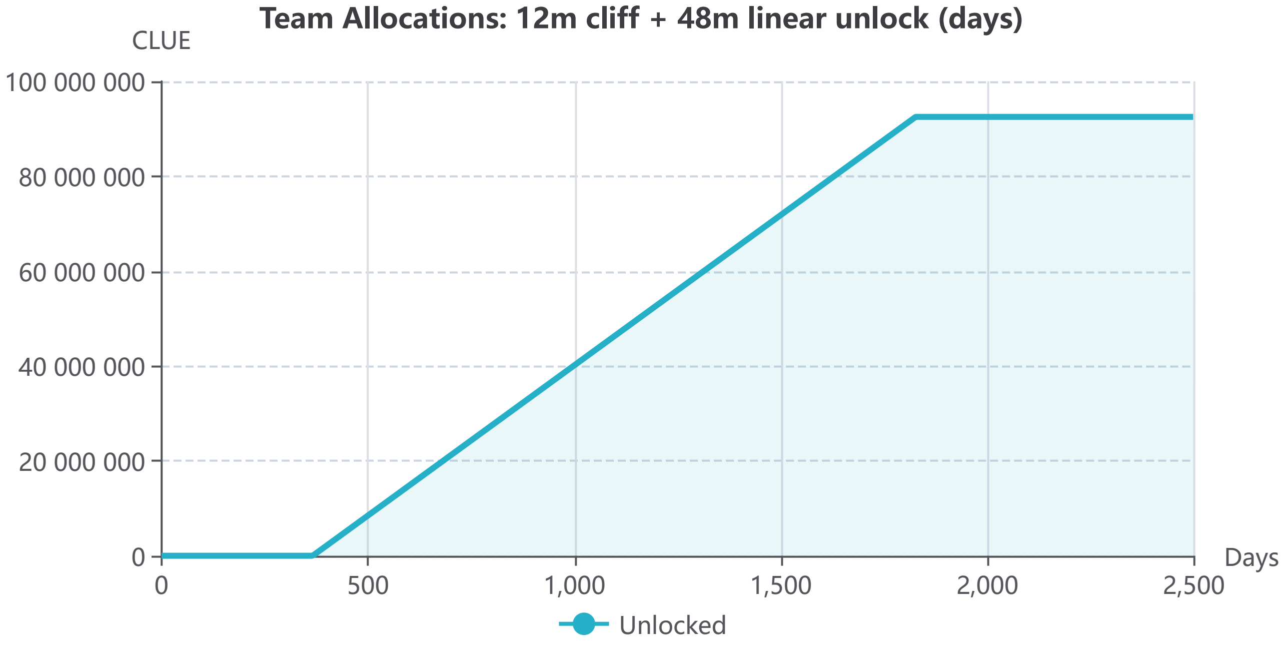
$$\text{Unlocked}(d) = \begin{cases} 0, & d < \text{cliffDays} \\ \text{Total} \times \min\left(1, \dfrac{d - \text{cliffDays}}{\text{linearDays}}\right), & d \geq \text{cliffDays} \end{cases}$$

  where d is the number of days since TGE.

- **Available volume:**

$$\text{Available}(d) = \max\big(0,\ \text{Unlocked}(d) - \text{Allocated} - \text{Staked}\big)$$

Result: Team Allocations in `DAOPool` create a long-term model for team participation in protocol governance. The annual cliff and four-year vesting enforces distribution discipline, strengthens the stability of governance layers and synchronizes the interests of the team with the long-term development of CLUE.



4.3.6. Operational Reserve

Volume: 7 500 000 CLUE

Unlock: Instant at TGE

Operational Reserve is the protocol's initial operational pool, designed to ensure continued operation of CLUE in the initial phase after TGE. Its function is to overcome the “cold start” problem and ensure the readiness of key ecosystem mechanisms at the time of launch.

Contract note: in the current contracts package there is no dedicated Operational Reserve contract with fixed unlock constants; this block is described as an allocation/governance policy.

Purpose

Operational Reserve is used to cover priority tasks of the early stage, where execution speed and predictability of the protocol are critical. The pool supports the launch of trading activity, primary roles and support operations until the system reaches a stable operating mode.

- **Launch of market activity:** providing the initial trading scenarios, including initial markets and initial user activity.
- **Role infrastructure support:** bootstrapping early-stage moderator participation, arbitration processes and related on-chain mechanics.
- **Initial ecosystem incentives:** financing limited early user/partner acquisition and growth programs.
- **Operational launch stability:** covering critical technical and service costs associated with the launch and stabilization of the protocol.

Spending policy

- **Unlock:** Full access at the time of TGE to ensure operational readiness of the protocol.
- **Control:** spending follows the approved governance policy with mandatory transparency of spending categories/use cases.
- **Pool balance:** the unused portion after stabilization of the launch can be redistributed by decision of the DAO to DAO Treasury or into liquidity pools.

Result: Operational Reserve ensures a controlled and technically robust launch of CLUE, where key protocol features are operational from day one without delays, and a limited pool size reduces the risk of excess market pressure.

4.3.7. Distribution Forecast

Total supply: 500 000 000 CLUE

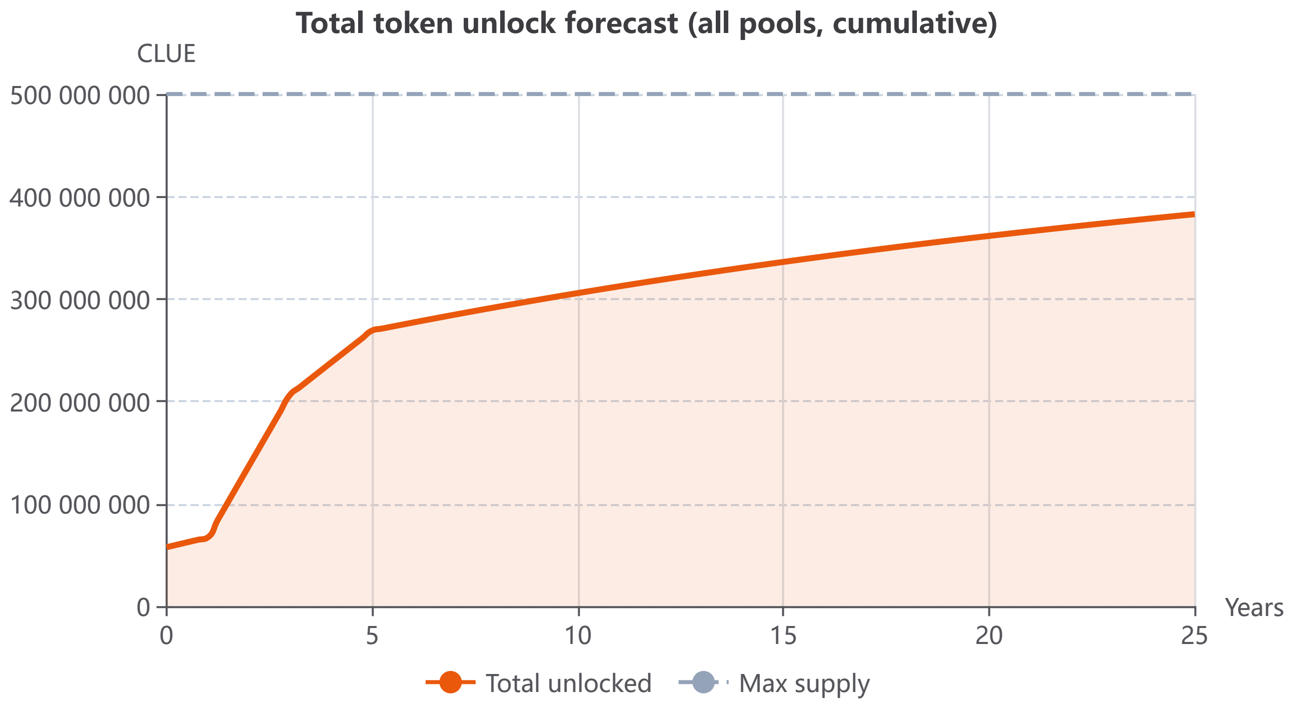
Model: Transparent unlock rules for pools

CLUE's distribution model is structured as a predictable, phased release system in which each pool has its own function, access restrictions, and verifiable unlock schedule. This structure reduces the risk of sudden supply spikes, maintains the stability of market liquidity, and forms a long-term link between protocol activity and token value.

Allocation logic

- **Infrastructure pools:** Liquidity & MM and DAO Treasury They use geometric unlock, which ensures a smooth token release rate from pre-allocated supply and predictability of development budgets.
- **Long-term stakeholders:** Private Round and Team Allocations are subject to cliff and linear vesting, which aligns the motivation of key stakeholders with CLUE’s long-term growth cycle.
- **Public distribution:** Public Distribution creates broad primary distribution and market price discovery on transparent terms.
- **Initial operational phase:** Operational Reserve ensures the protocol is ready to launch on TGE day and covers critical early stage tasks.

Taken together, this model creates a balanced distribution curve where short-term launch needs are combined with long-term token release discipline. CLUE avoids aggressive early release scenarios and builds sustainable tokenomics that support ecosystem development throughout the protocol's lifecycle.



4.4. Token Smart Contracts

The CLUE token architecture is modular: issuance, deflation, distribution and governance are separated into independent contracts. This approach simplifies auditing, reduces the risk of privilege concentration, and allows the protocol to scale robustly across networks without changing the underlying economic logic.

Governance and upgrades

- **Owner = DAO:** key contracts belong to the DAO and are executed through DAO governance procedures (and optional executor routing); the team does not have direct administrative privileges.
- **Managing module addresses:** changes in critical addresses and module connections occur through an on-chain procedure and are publicly recorded before use.
- **Token and pools parameters:** unlock rates, limits and protection modes are changed only by governance decisions of the community.

Security invariants

- **Fixed supply:** Additional mint operations are not supported.
- **Irreversible burn:** burning reduces supply and is fixed on-chain.
- **Predictable unlocks:** geometric decay unlock schedules reduce the risk of supply surges.
- **Full traceability:** movements between pools and treasury are logged by events and are publicly auditable.
- **Cross-chain adapters:** are connected separately and activated only by decision of the DAO, which preserves a single supply model and eliminates hidden emissions in third-party networks.

As a result, CLUE's smart contract architecture forms a reproducible and verifiable token system, where the rules for issuance, deflation and distribution are executed by code and do not rely on manual operational management.

4.5. Protection Against Concentration and Scam Risks

Protection against concentration and scam in CLUE is designed as a multi-layer system that combines tokenomics, governance procedures and on-chain moderation and risk-labeling layer. All key parameters are fixed in smart contracts and can only be changed through DAO governance delays (grace + executionWindow), which eliminates opaque manual interventions.

Anti-concentration at the token level

- **Fixed supply:** no additional minting is possible; scarcity is amplified through deflationary tokenomics.
- **Smooth unlock:** Liquidity & MM and DAO Treasury are unlocked geometrically, without cliff unlocks and price shocks.
- **Long lock periods:** For sensitive allocations, long-term restrictions and controlled token utility routes are applied.
- **Limited operating reserve:** reserve spending is governed by the DAO procedures and on-chain control.
- **Early-stage limits:** DAO may include time restrictions, including per-wallet cap, cooldown and other anti-dumping parameters at an early stage of liquidity.

Governance safeguards

- **Default Timelock:** changes to fees, addresses and modes are not applied instantly and are published in advance.
- **Limiting the influence of passive token accumulation:** The governance model takes into account not only volume, but also participation in the protocol system.
- **Controlled layers of discounts and roles:** parameters for access to economic privileges are set on-chain and cannot be monopolized outside the DAO rules.

Anti-scam market framework

- **Paid listing and moderation:** a basic launch fee and stake-backed moderation eliminate spam and low-quality markets.
- **Risk labeling:** moderators mark markets by risk level, including scam risks and low-quality, which reduces the proportion of problematic launches.
- **Appeals and Arbitration:** disputes are resolved on-chain through a formalized dispute process with deposits and financial stakes.
- **FX-hedge execution controls:** when FX mode is enabled, AMM routes the trading-token ↔ stable hedge through `MarketAMMFXTreasury` with registry checks, pool defaults, and slippage boundaries (`minOut`/`maxIn`), reducing manipulation vectors during volatile legs.
- **Synthetic NO integrity:** `MarketAMMNoAggregator` allows NO-side flow only with complete YES-leg coverage (`N-1` outcomes), deduped outcomes, equal leg sizes and payout checks, which blocks “partial basket” scam patterns.
- **Slashing mechanics:** Erroneous or dishonest actions by moderators and arbitrators lead to penalties and limit incentives for abuse.

Transparency and monitoring

- **Public addresses and events:** pools, treasury and key transfers are available on-chain and are reflected in dashboards.
- **Liquidity reporting:** transfers to DEX and CEX pools are accompanied by disclosure of tx-hash and volume.
- **Anomaly alerts:** Large withdrawals and atypical position changes can trigger on-chain alerts/monitoring signals.
- **Event log:** public feed of ecosystem activities, including listings, partnerships, pool movements, and DAO voting results.

Together, these mechanisms limit concentration through predictable unlock schedules, staking mechanisms and governance procedures, and the anti-scam layer improves the quality of markets and reduces the risk of manipulation. As liquidity grows, the parameters can be refined, but only through an on-chain decision by the community.

5. Staking and Incentive Mechanisms

Staking in CLUE is a multi-level on-chain system where the same CLUE token is distributed between three functional pools: **moderation/arbitration**, **fee-discount** and **governance**. Each pool has its own rules for entry, rotation, reward/penalty and exit, but they are all connected by a single economic cycle of the protocol.

Key principle - **dynamic competition for slots**. If active slots are filled, a new participant is included in the active set only if they exceed the threshold of the weakest participant (by stake or by effective weight depending on the pool). This makes the system dynamic: influence cannot be "bought once and for all"; it must be constantly confirmed by stake and activity.

5.1. Moderator and Arbitrator Staking

The quality-control pool operates through `ModeratorsPool` and `ModeratorsManager`. Here the stake determines access to committees, the probability of selection, the reward share and the severity of slashing for mistakes.

5.1.1. Pool model

- **Two-tier structure:** the pool maintains *top* (active) and *reserve* lists.
- **Roster limits (current profile):** active participants - 50, reserve - 10.
- **Committees:** moderation committee — 3, arbitration committee — 5.
- **Weight limit per participant:** moderation up to 49%, arbitration up to 40% (anti-concentration in committees).

5.1.2. Effective weight, activity and slashing score

- **Effective weight of the participant:** `effectiveWeight = activeStake × (activityBps + slashingBps) / (2 × 100000)`. Weight depends not only on stake volume, but also on decision quality.
- **Activity score tracking:** reward/penalty steps are set by the manager; in the current configuration typical steps: moderation +5% / -10%, arbitration +1% / -15%.
- **Slashing score tracking:** reflects the quality of decisions and affects the final effectiveWeight; typical steps: moderation -15% penalty / +5% recovery, arbitration -30% penalty / +1% recovery.
- **Result:** even with the same stake, the participant with better discipline receives more weight, enters the active set more often and retains a place in the active set more consistently.

5.1.3. Token Slashing (Stake Confiscation)

In addition to score penalties, in severe cases direct DAO-slashing is used via `slashStake(...)`. The fine can be applied in batches to several addresses with individual shares.

- **Calculation basis:** The slashable balance includes the active stake and the participant's pending unbonding queue.
- **Formula:** `slashAmount = slashableBalance × shareBps / 100000`.
- **Destination of slashed funds:** the confiscated amount is transferred to the recipient approved by the governance decision.
- **Consequence:** a lower stake automatically reduces effectiveWeight and accelerates displacement from the active set.

5.1.4. Moderator/Arbitrator Participation Cycle

a) **Stake:** the participant locks CLUE and receives an initial score profile.
b) **Membership:** the address enters the active or reserve roster according to the current effectiveWeight.
c) **Participation in rounds:** actions in moderation/arbitration change the activity/slashing score.
d) **Reward / penalty:** rewards strengthen the position, mistakes reduce weight and increase the risk of rotation.
e) **Withdrawal:** `requestUnstake → unbonding → claimUnbonded` (no instant withdrawal).

5.1.5. How participant replacement works

- **Entry threshold when the active set is full:** the new member must exceed the lowest active member threshold.
- **Promotion from reserve:** The best reserve candidate is promoted to active when its effectiveWeight is higher than the weakest active member.
- **Score degradation:** passivity/errors can demote the address to the reserve even without withdrawing the stake.
- **Slash effect:** reducing stake directly accelerates loss of seat and lowers the chance of re-concentrating influence.

5.2. Discounts on trading fees

The discount mechanism is implemented by the combination `FeeDiscountPool` + `FeeDiscountRegistry` + `FeeDiscountManager`. The user stakes CLUE and competes for seats in limited discount slots.

5.2.1. Pool model

- **Discount ladder:** 10 slots with DAO-controlled discountBps and capacity.
- **Total seat capacity (current profile):** 25 active seats.
- **Reserve:** 5 reserve seats for instant rotation when stakes change.
- **Slot matrix (default):** discounts `99%, 90%, 80%, 70%, 60%, 50%, 40%, 30%, 20%, 10%` for capacity `1,1,1,2,2,2,3,3,5,5`.

5.2.2. Entry, threshold and seat allocation

- **If seats are available:** the participant enters the active slot roster directly.
- **If the limit is full:** Entry is only possible under the strict condition `newStake > weakestStake`.
- **Real time recalculation:** every change in stake calls `notifyStakeChanged` and triggers top/reserve rebalance.
- **Maximum discount:** in the current profile - up to 99%, rather than the default 100%.
- **Effective fee:** `fee_eff = fee_base × (1 - discountBps / 100000)`.

5.2.3. Member lifecycle in fee-discount pool

a) **Stake in FeeDiscountPool.**
b) **Checking the entry threshold:** with a full seat-limit, you need to outbid the weakest seat.
c) **Slot assignment:** after rebalancing, the participant receives slotId and the corresponding discountBps.
d) **Trading on the AMM:** The effective fee is reduced, and the saved volume is reflected in account-level metrics.
e) **Unstake/exit:** requestUnstake → unbonding → claimUnbonded, after which the slot can be released and occupied by another address.

5.2.4. How displacing works in slots

- When active slots are occupied, the new candidate must stake above the active slot's weakest threshold (strictly `>`).
- As the stake grows, the participant can move up to a higher slot (with a larger discount), automatically displacing the address with a lower priority.
- In the event of a stake drop/partial withdrawal, a participant may be demoted to a lower slot or pushed into reserve if larger stakers bypass it.
- If a trader does not have a seat in the top-slot set, their discount becomes zero until they re-enter the active slot roster.

5.2.5. Cliff/Vesting in the discount pool

FeeDiscountPool supports the same lock mechanisms as other staking pools: `minWithdrawTimestamp` (cliff) and `vestingPeriod`. This allows you to receive a discount in the active phase, but release the stake gradually and predictably for the market.

5.3. Staking for protocol governance

The governance mechanism is built on `DAOPool` + `DAORegistry` + `DAOManager`. A stake in DAOPool gives vote weight, but the final influence is calculated based on activity in actual voting.

5.3.1. DAO staking model

- **Active set (current profile):** 15 participants.
- **Reserve:** 5 participants.
- **Effective weight:** `effectiveWeight = activeStake × activityBps / 100000`. Passive stake loses influence faster than active stake.
- **Proposal snapshot:** at the time of proposal creation, the composition and weights are recorded; this blocks after-the-fact manipulation of the roster.
- **DAO Rewards:** are distributed proportionally to effective weights in the active set, not only by raw stake volume.

5.3.2. participation rate and impact on weight

- **Reward for participation:** **+25 percentage points** to activity (up to max).
- **Penalty for missing:** **-50 percentage points** (not lower than min).
- **Economic result:** decreasing activity reduces effectiveWeight and the share of DAO reward distribution.

5.3.3. Participant governance cycle

a) **Stake in DAOPool** and entering the active/reserve roster based on current thresholds.
b) **Proposal snapshot:** At proposal creation, the available voting weight is recorded.
c) **Voting:** participation increases participation score; missed votes reduce it and, therefore, future effective weight.
d) **Finalization/execution:** past decisions are executed in the governance window, and the inactivity penalties are reflected in the score model.
e) **Unstake:** exit from the governance pool follows unbonding and delay, which reduces the risk of an “instant exit after voting”.

5.3.4. How positions are reallocated in DAO

- If the active-limit is full, the new address becomes active only after the current entry threshold has been overcome.
- A participant from reserve can displace the weakest active member when its effectiveWeight becomes higher.
- Systematic abstaining from voting reduces activityBps and automatically weakens influence without forcing the stake to be withdrawn.

5.3.5. Cliff/Vesting in the governance pool

DAOPool supports cliff and vesting lock configurations for phased exit. This allows the holder to participate in governance over a long horizon, while reducing the risk of a sharp loss of voting weight after key votes.

5.4. Interplay of Three Pools: A Unified Competitive Model

The three CLUE staking pools form a common capital dynamic: the same token competes for different utility rights - **role** (moderation/arbitration), **savings** (fee discount) and **influence** (governance). Therefore, participants constantly redistribute stake between pools depending on expected profitability and strategic goals.

- **Pools compete for stake through yield/reward attractiveness:** reward growth in moderation pulls part of the capital from discount slots. The growth of trading volume and fees increases the demand for discount slots.
- **The DAO manages the loop parameters:** top/reserve limits, activity steps, unbonding and slot configurations are changed by a governance decision, and not by manual centralized intervention.
- **Rotation instead of static dominance:** when the limits are filled, the influence remains only while maintaining the stake and activity. Passive addresses are replaced by active ones.
- **Governance capture protection:** In each subsystem there is a mechanism for threshold entry, reserve rotation and disciplinary coefficients, so control is distributed dynamically and is not permanently fixed.

Section summary: staking in CLUE is a continuous loop **stake** → **participation** → **score/weight recalculations** → **roster rotation** → **reward/penalty** → **restake**, where each pool has its own competition mechanism, but all three operate as a single protocol economic system.

5.5. Staking types: cliff and vesting

In all three CLUE staking pools (`ModeratorsPool`, `FeeDiscountPool`, `DAOPool`) the lock profile is set at the level of each stake record (`StakeRecord`: amount, minWithdrawAt, vestingPeriod, withdrawn). Key point on contracts: **cliff and vesting are applied upon entry, but also upon exit** via `requestUnstake`, where the volume is broken down by the FIFO history of the stakes.

The resulting `minWithdrawAt` (cliff) is calculated as the maximum between the user parameter and the global pool manager delay (`now + minWithdrawDelay`/`minWithdrawTimestamp`). Therefore, even a “basic” stake is subject to the system’s minimum withdrawal window.

5.5.1. Type A — Basic Stake (Without Individual Vesting)

- **Entry:** stake without individual `vestingPeriod` (`vestingPeriod = 0`).
- **Cliff:** The individual timestamp does not need to be specified, but the global delay of the pool manager is in effect.
- **Withdrawal:** after unlocking `requestUnstake` transfers the volume to the unbonding queue (without creating a vesting-schedule).
- **Claim:** After the unbonding period, tokens are collected via `claimUnbonded`.
- **When to use:** when you need a classic `lock → unbond → claim` sequence without an extended release schedule.

5.5.2. Type B — Cliff-Based Stake (Delayed Withdrawal)

- **Entry:** stake with `minWithdrawTimestamp` (cliff date).
- **Cliff normalization:** `enforcedTimestamp = max(minWithdrawTimestamp, now + globalDelay)` is used in the contract.
- **Until cliff:** the exit request is limited by the lock period (`LockedBalance`).
- **After cliff:** the usual loop `requestUnstake → unbonding → claim` works.
- **Important:** in `DAOPool`/`FeeDiscountPool` timestamp must be strictly in the future; in `ModeratorsPool` the boundary value is automatically reallocated by at least +1 second.
- **Effect:** protection from “immediate entry/exit” and a more predictable composition of active participants.

5.5.3. Type C — Stake with Cliff + Linear Vesting

- **Entry:** stake with two parameters: `minWithdrawTimestamp` (cliff) and `vestingPeriod`.
- **When/where vesting is created:** not in `stake`, but in `requestUnstake` when processing records with `vestingPeriod > 0`.
- **Schedule parameters:** for each such portion, `VestingSchedule(total, claimed, cliff, start, end)` is formed, where `start = max(nowTs64, cliff)`, `end = start + vestingPeriod`.

- **Interaction with unbonding:** in one `requestUnstake` can simultaneously appear *toUnbond* (for records without vesting) and *toReserve* (for records with vesting).
- **Reserve mechanics:** vesting volume is taken into account as `vestingReserved` and cannot be withdrawn again by a second order.
- **When active stake decreases:** for vesting portions, debiting from `active` occurs when the actual claim is made (in `claimUnbonded`), and not at the time of creating the schedule.
- **Effect:** smooth exit from a position without sudden pressure on pool composition and the economics of roles/discounts/governance.

5.5.4. How it works step by step (general life cycle)

a) **Stake:** the participant contributes CLUE to the selected pool.
b) **Recording:** `amount/minWithdrawAt/vestingPeriod` are stored in `stakeHistory`.
c) **Active phase:** stake participates in rankings, discount slots, or governance weight.
d) **Check withdrawable:** `requestUnstake` allows withdrawal only within the limits of `active - lockedActive - vestingReserved` (in moderators the slashed part is additionally taken into account).
e) **FIFO partition:** the system iterates through stake history in order and splits the amount into an unbonded portion and a vesting portion.
f) **Claim:** `claimUnbonded(maxItems)` processes both queues (unbond + vesting) up to the limit `maxItems` in one call.

5.5.5. Linear vesting formula

The contracts use piecewise logic: **`if t ≤ start: claimable = 0`**; **`if t ≥ end: claimable = total - claimed`**; otherwise **`claimable = floor(total × (t - start)/(end - start)) - claimed`**.

Where `start = max(nowTs64, cliff)`. Before start, 0 is available, then the amount unlocks linearly and is fully claimable by `end`.

Practical summary of the code: **cliff** limits the start time of the withdrawal, **unbond** sets the minimum delay for regular chunks, and **vesting** stretches release for selected portions. The combination of these three layers makes the exit from the stake predictable and reduces the risk of sudden drawdowns in active pool compositions.

6. Governance

Governance operates as an open decentralized system: the community stakes tokens, puts forward ideas, votes and executes decisions directly through on-chain processes. The goal is for any changes to the protocol, budgets and parameters to take place transparently, without hidden admin controls and manual edits.

All logic is embedded in smart contracts and is visible on the blockchain: the treasury unlocks smoothly according to a formula, rights and limits are set by voting, and any action leaves an on-chain record (event log). This makes management predictable for participants and resistant to human error.

6.1. DAO model

- **Active set:** determined by effective weight (`stake × activity%`) for up to 15 addresses. When the limit is full, you can only get into the active set by staking more than the weakest participant.
- **Reserve and dynamic minimum:** when the limit of active slots is full, entry is possible only with a stake higher than the current weakest. In parallel, a reserve list is maintained for automatic roster rotation.
- **Weight of votes and rewards:** **`stake × activity%`**; starting activity - 100%, range 5–100%. The voting power snapshot is captured when the proposal is created. Only addresses from this snapshot can vote and receive penalties/rewards.
- **Activity for actions:** every vote increases activity by +25 percentage points (but not higher than the maximum); a missed vote reduces it by -50 percentage points, but not below the minimum.
- **Voting duration:** manager-configurable within on-chain bounds in `DAOManager` (the author chooses a value within current limits).
  For emergency situations, a separate whitelist of addresses is provided, which can create emergency proposals (for example, a protocol pause or a critical parameter fix) with a shortened minimum voting period configured in `DAOManager`.
  Even in emergency mode, quorum, weight snapshot, grace and executionWindow are preserved.
- **Default quorum:** 50% from snapshot weight; the author can raise the threshold to 100%. Works only with a positive quorum.
- **Decision criterion:** **`Yes > No`** when quorum is met; tie or quorum not met = rejected.

- **Signaling proposals:** allowed without a target call (target = 0, calldata empty) - record DAO intent without on-chain execution of the action.
- **Finalization and execution:** after the deadline the grace period begins (creator-only finalization), then executionWindow opens (any address can finalize), and the action is executed automatically at the moment of finalization. Grace and executionWindow durations are manager-configurable. After the window, anyone can call `cancelProposal`; if the proposal passes, penalties for non-participation and non-execution apply.
- **Submission mode:** By default, only addresses from the whitelist can submit. By DAO voting you can choose: *Whitelist only* - only pre-approved addresses; *DAO only* — any current members with non-zero weight; *All* - any address with valid action payload. Mode switching is recorded on-chain.
- **Governance capture protection guard:** if one participant gains ≥ 50% of active weight, the *All* mode is automatically narrowed to whitelist-only for new proposals.
- **Stake withdrawal:** unbonding request and minimum withdrawal delay are manager-configurable in `DAOManager`.
- **Fees from trades without referral:** 25% collected fees (0.75% from the transaction at the base rate 3%) is sent to the DAO and distributed among active participants in proportion to their effective weight.
- **Paid usernames as additional DAO revenue:** paid user aliases (the cost of one username is 10 000 CLUE) generate additional income for DAO Treasury and enhance the value of membership.
- **DAO Governance Scope:** DAO has complete control over the protocol: from changing parameters and turning services on/off to replacing contracts through voting and timelocking. DAO Treasury budgets and spending are also approved and reconfigured by community votes.

6.2. Governance Roles

- **Author of the proposal:** generates CIP with metadata and precise action; submission is available in the selected DAO mode (whitelist , dao only or all), which is enabled by voting.
- **Delegation:** delegation is not used in the current on-chain voting model.
- **Voters:** participate in voting with snapshot weight from DAOPool at the time of proposal creation.
- **Finalizers/executors:** after the grace period, any address can finalize the proposal within the executionWindow; there is no separate execution step - the action is performed during finalization.
- **Whitelist / emergency submitter:** addresses from the whitelist can create emergency proposals with a shortened minimum voting period; they do not receive voting rights beyond the stake.

6.3. Treasury management and proposal/voting process

CLUE Governance is an **on-chain governance process**, which turns community decisions into executable actions in smart contracts. Participants stake CLUE → create a proposal → vote → record the result → perform an action → move funds from the treasury if necessary.

Basic route: **new proposal** → **vote** (within vote duration) → **grace** (finalization by creator only) → **executionWindow** (finalization by any address) → after window **cancelProposal**.

Important: There are no hidden controls in CLUE and the project does not depend on the creators, who can change the rules or withdraw the treasury instantly. Any change in parameters, addresses or access rights goes through the same public on-chain path.

Full lifecycle: from proposal to treasury execution

- **Preparation.** The participant stakes CLUE in the DAO Pool. When a proposal is created, the registry records the weights of the current active DAO members and stores them in voting slots.

  Important: the snapshot is recorded only for the active set (top-N); addresses outside this set do not vote on the proposal.

  Practical implication: the composition of voters and their “weight” is fixed at the time of start, so that the balance of power cannot be changed after the fact.
- **Creating a proposal.** The author submits a request with a reference to the description (CID), specifies the address of the contract to be called, and the data for this call. The voting duration is selected within acceptable limits: from the minimum allowed to the maximum allowed; if the author does not specify anything, the default period is applied. The submission mode (whitelisted addresses only / DAO stakers only / everyone) determines who can submit proposals.

  Target must be a contract if there is calldata; signal proposals are allowed (target = 0 and calldata is empty) without on-chain execution of an action.

From this point on, a public proposal ID and description appear and can be checked in the explorer. The logs also record a hash of the transmitted action data so the community can verify that the exact action payload put up for discussion is what gets executed.

- **Cancel proposal.** Before finalization, the author (or owner of the manager) can cancel the proposal to eliminate erroneous or outdated actions.
- **Voting.** The weight is distributed between "Yes" and "No" according to the pre-recorded snapshot. Each vote is published with a link to the rationale; those who vote receive an increase in their activity score, and exceeding the available weight is not allowed.

  You can vote in parts, rather than "all or nothing." Each vote is recorded in the log: who voted, how and with what amount of weight, as well as how much weight has already been used.
- **Finalization.** After the deadline, the creator can finalize the proposal in the grace period; after grace, anyone can finalize within the executionWindow. Quorum is calculated by snapshot, the outcome is recorded by votes. Finalization immediately executes the action (if target is specified) and records the execution status.

  If the proposal passes, penalties for non-participation are automatically applied upon finalization.

  After finalization, you cannot "re-vote": either the decision is executed at the time of finalization, or the window will be missed and a new proposal will be required.
- **ExecutionWindow.** During executionWindow, any address can finalize the proposal if the creator has not done so during grace. The action is performed within finalization; if execution is not initiated by the creator (after grace), a penalty for non-execution is applied to the creator.
- **Cancel after window.** After executionWindow has expired, anyone can call `cancelProposal`. If the proposal passes by vote, penalties for non-participation and for non-execution by the creator are automatically applied.
- **Inactivity penalties.** Penalties for non-participation are applied automatically when finalizing a previous proposal (outcome Yes) and when canceling after the window. This reduces the effective weight of passive holders and keeps the DAO composition active.
- **Movement of treasury funds.** After a successful decision, the owner of the treasury makes a withdrawal to the agreed address and amount within the unlocked limit; the unlocking itself occurs automatically according to the formula and does not require manual intervention.

  The treasury is being unlocked gradually: only tokens in withdrawable balances are available. An attempt to take more is rejected, and the withdrawal is recorded in the logs.
- **DAO fee share.** When unreferred trading fees or other payments accrue to the treasury, the registry immediately divides them among active participants according to current weights and records the distribution in the logs.

  The distribution is transparent: the event shows the token, amount, number of recipients and total weight.

  Accruals go into pending status, and participants collect them via on-chain `claim`.

The emergency path (if enabled) is limited to whitelisted addresses and uses the same basic principles: snapshot, verified quorum, grace and executionWindow. In emergency mode, a shortened minimum voting period is allowed, but quorum, snapshot, finalization rules and automatic penalties are not bypassed.

How it works with contracts (explained step by step)

Briefly linking steps to specific contracts and events.

- **DAORegistry** (`createProposal` / `vote` / `finalize` / `cancelProposal`): when submitting, captures a snapshot of the weights of top participants at the limit `activeMembersLimit`, validates target/code/calldata and allows signaling proposals when `target = 0`. Submission modes (Whitelist/DaoOnly/All) are applied on-chain; when one participant dominates (≥ 50% effectiveWeight) in the mode *All* submission is limited to whitelist. For whitelisted addresses, an emergency minimum voting duration is allowed (via `emergencyMinVoteDuration`), without bypassing quorum and snapshot.

  Finalization: only the creator can finalize during the grace period; after grace, any address can finalize within the executionWindow. Finalization immediately tries to execute the action (if target is specified) and records the execution status (success/reverted).

  Penalties for non-participation are applied automatically when a proposal passes; if finalization is not performed by the creator after grace, the creator receives a penalty for non-execution. After executionWindow has expired, anyone can call `cancelProposal`; if outcome = Yes, penalties are applied automatically.

- **DAOManager**: stores the DAO configuration: voting duration boundaries, `defaultVotingPeriod`, `defaultQuorumBps`, `resolveGracePeriod`, `executionWindow`, separate emergency minimum voting period for whitelist addresses, active participant limit and reserve (`activeMembersLimit`, `extraReserveStakers`), activity steps and their range, as well as unstake/withdrawal parameters. All changes are available only to the owner (DAO governance) and are enacted through on-chain decisions.
- **DAOPool**: Governance role staking. Stores `stake`, counts `effectiveWeight = stake × activity%`, a snapshot of this weight is included in the proposal. When the limit is full, a dynamic minimum applies: in order to enter the active set, a new participant must exceed the weakest. A standby queue is maintained. Unstaking follows a request and a queue, with delays specified by the DAO. Active voters receive an increase in activity by calling `rewardActivity`, missed participation is recorded automatically when finalizing/cancelling past proposals.

  Practical result: those who vote systematically increase their activity% and their effective weight; passive holders lose voting power and share of rewards.
- **DAORewardsDistributor** → **DAORegistry**: unallocated referral fees and other fees/rewards are sent to the distributor, which forwards them to the registry. It calls `distributeDaoRewards` and divides the amount in proportion to current active-set effective weights at distribution time (within `activeMembersLimit`). Accruals are recorded as pending and are collected by participants via `claimDaoReward`. All counters are written to `ClueAccounts` and logged by the event `DaoRewardsDistributed` with the fields token/amount/number of recipients/total weight.
- **DAOTreasury** — a separate vault of the CLUE token with a mathematical unlock schedule.

  Every **24 hours**, **0.008%** of the remaining balance is automatically released (constant **`RELEASE_RATE_BPS = 8`** in the contract; **≈2.9%** per year according to the formula $1 - (1 - 0.00008)^{365}$).

6.4. DAO Governance Mechanics

The mechanics of DAO governance are designed so that influence is calculated by stake and activity, the DAO remains active and protected from passive holders, proposal submission modes can be flexibly chosen, and incentives encourage participation.

Our goal is to combine two things: **security** (treasury and parameters are protected by procedure and limits) and **governance agility** (the community can actually change the protocol without waiting for a central team). That is why the mechanics include both incentives (rewards for active participants) and safeguards (quorum, execution windows, treasury restrictions, penalties for non-participation).

Voting Power: Calculating Effective Weight

In CLUE, not only the size of the stake is important, but also participation in the life of the DAO. Therefore, the effective weight is used: **`effectiveWeight = stake × activity%`**. This means that two participants with the same stake can have different real weights if one votes consistently and the other systematically ignores proposals.

- **Stake** — the actual staked amount of CLUE tokens in `DAOPool`.
- **Activity%** — activity coefficient in the range **5–100%**, which varies from participation/non-participation.
- **Active set (top-N).** For key decisions and distributions, a limited set of the most significant participants is used (parameter `activeMembersLimit`).

Submitting proposals: who can initiate changes

The protocol provides submission modes so that the DAO can choose the level of openness at different stages of development:

- **Whitelist** - only pre-authorized addresses can create proposals (for example, for "emergency" or for an early stage).
- **DaoOnly** — DAO members (stakers with voting rights) can create proposals.
- **All** — any address can create a proposal if it is valid according to the rules (target/call).

The mode is not a “governance parameter”, but an on-chain parameter that is updated by the DAO procedure (see `ProposalSubmitModeUpdated` and `DaoWhitelistUpdated` events in `DAORegistry`).

Voting Discipline: The Cost of Abstaining

Governance breaks down when large stakers don't participate (or only participate when they need to). Therefore, CLUE provides transparent discipline mechanics:

- **Reward for participation.** When voting, `rewardActivity` is called in `DAOPool`, increasing activity%.
- **Penalty for non-participation.** Penalties are calculated automatically when a previous proposal is finalized (outcome Yes) or when canceled after the window; there is no longer a separate non-voter penalty function call.
- **Effect.** Over time, the passive participant loses effectiveWeight, and therefore the influence and share of the DAO income distributions.

Capture protection and emergency modes

- **Top-N + reserve:** voting is performed only by active participants (top-N), the reserve ensures rotation and reduces concentration.
- **Dynamic entry minimum:** when all active slots/seats are filled, the new staker must surpass the weakest active participant, which reduces the risk of diluting influence with small addresses.
- **Governance capture protection guard:** at ≥ 50% effectiveWeight in one participant, submission mode *All* is limited to whitelist.
- **Timelock delay:** There is a grace period before execution, which gives time for the community to react.
- **Penalty for non-execution:** if the creator did not finalize the proposal by the end of the grace period and another address finalized it (or the proposal was canceled after the window), the creator receives an automatic activity penalty.
- **Emergency mode:** whitelisted addresses can run proposals with a shortened minimum voting period, but without quorum bypass or snapshot.
- **Unbonding/withdraw period:** exit from a stake is not instantaneous and is performed with delays, which reduces the risks of sudden exit after voting.

Treasury: why it cannot be “drained in a single vote”

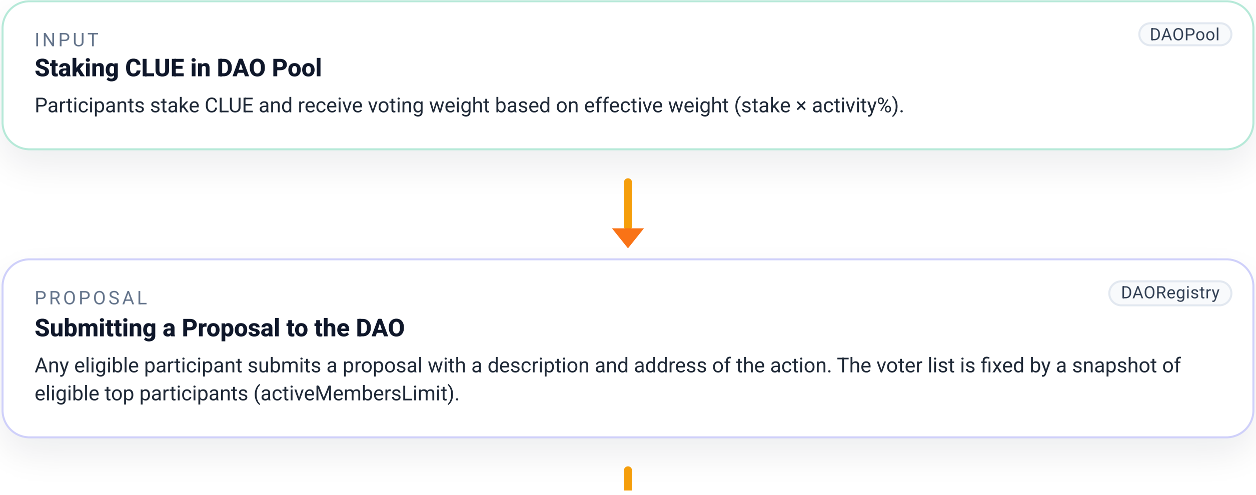
The treasury `DAOTreasury` is protected not only by the voting procedure, but also **by a mathematical limit on the withdrawal rate**. Even if the DAO decides to make a large withdrawal, the contract will not physically allow you to withdraw more than what has already been unlocked according to the formula. This eliminates the scenario of a single vote instantly draining the treasury.

Unlocking works as follows: **geometric decay unlock schedule**, every **24 hours**, **0.008%** of the remaining balance is automatically released (rather than a fixed percentage of total). Therefore, the rate of release decreases over time: at the beginning, more is released in absolute terms, then less and less.

Any withdrawal goes through a limit check: `withdraw(to, amount)` compares `amount` with what `availableToWithdraw()` returns. If you try to withdraw more than what is available, the transaction will be rejected. So the DAO manages *direction* and *allocation* of funds (to whom/where/why), but cannot speed up the unlock from the treasury beyond the predefined formula.

DAO Governance Diagram

Stake → Snapshot → Proposal → Voting → Execution → Treasury Withdraw



INPUT

DAOPool

Staking CLUE in DAO Pool

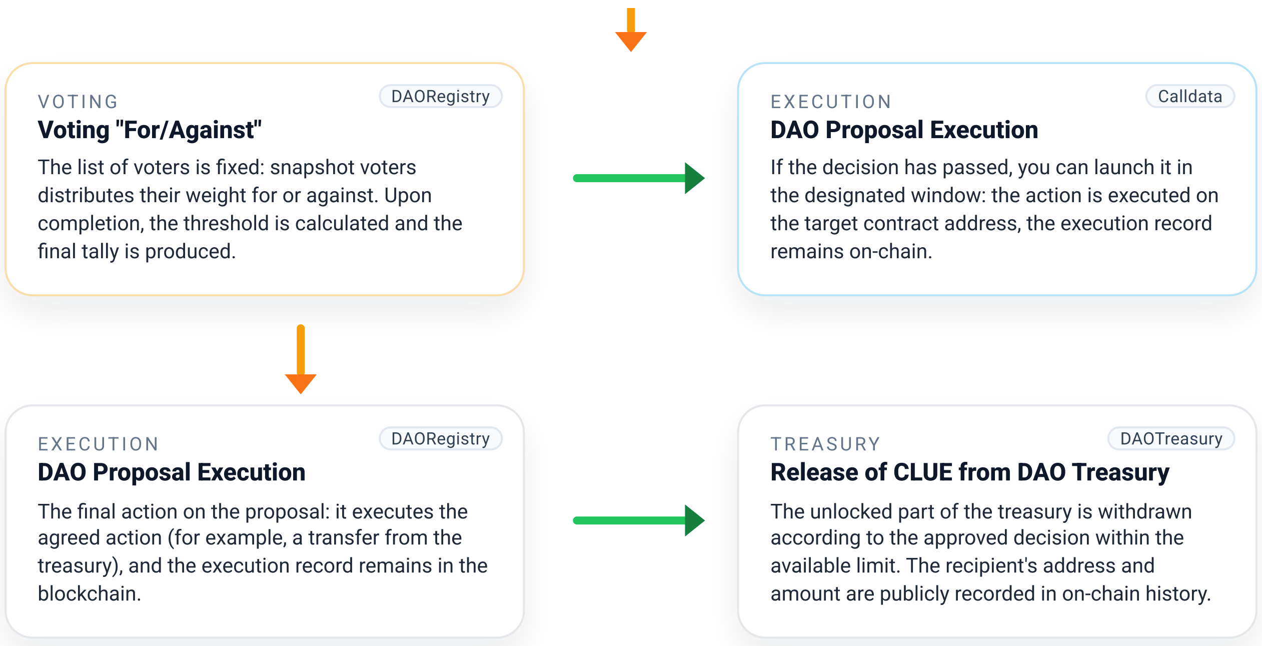
Participants stake CLUE and receive voting weight based on effective weight (stake × activity%).

PROPOSAL

DAORegistry

Submitting a Proposal to the DAO

Any eligible participant submits a proposal with a description and address of the action. The voter list is fixed by a snapshot of eligible top participants (activeMembersLimit).



VOTING

DAORegistry

Voting "For/Against"

The list of voters is fixed: snapshot voters distributes their weight for or against. Upon completion, the threshold is calculated and the final tally is produced.

EXECUTION

Calldata

DAO Proposal Execution

If the decision has passed, you can launch it in the designated window: the action is executed on the target contract address, the execution record remains on-chain.

EXECUTION

DAORegistry

DAO Proposal Execution

The final action on the proposal: it executes the agreed action (for example, a transfer from the treasury), and the execution record remains in the blockchain.

TREASURY

DAOTreasury

Release of CLUE from DAO Treasury

The unlocked part of the treasury is withdrawn according to the approved decision within the available limit. The recipient's address and amount are publicly recorded in on-chain history.

Technical contract architecture

Technical contract architecture: how modules are connected and which component is responsible for its function. All key contracts in the CLUE ecosystem are governed through the DAO governance loop (owner routing in practice is configured through DAORegistry/DAOExecutor and, if necessary, timelock at deployment), so any changes, limits, and withdrawals go through DAO decisions.

DAORegistry - proposals, voting, execution

- **Creation:** `createProposal(metadataCid, target, actionCalldata, votingPeriod, quorumBps)` checks the validity of the action (target/code/calldata), duration boundaries and correct quorum. The voting power snapshot is captured by top-N, and the parameters `resolveGracePeriod` and `executionWindow` are copied from `DAOManager`. Logs `ProposalCreated` (including `actionCalldataHash`).
- **Voting:** `vote(id, choice, weight, voteMetadataCid)` works only for addresses with snapshot weight; limits the total weight used and logs `ProposalVoted`.
- **Finalization and execution:** `finalize(id)` is only available after `votingDeadline` and before `executionDeadline = votingDeadline + grace + executionWindow`. During the grace period, only the creator can finalize it; after grace - any. Finalization checks quorum, sets the outcome, and automatically applies penalties for non-participation if the proposal passes. If `outcome = Yes` and target is specified, the call is executed immediately; the result is recorded in `ProposalActionExecutionResult`, and `executionStatus` takes the value success/reverted. If the execution was not performed by the creator after grace, the creator receives a penalty for non-execution.
- **Cancel:** `cancelProposal(id)` is available to the creator/owner until executionWindow expires. After the window, any address can cancel. If `outcome = Yes`, penalties for non-participation and an additional creator penalty for non-execution are automatically applied.
- **Submission mode and whitelist:** `setProposalSubmitMode`, `setDaoWhitelist` (available to the manager owner) log `ProposalSubmitModeUpdated` and `DaoWhitelistUpdated`.

DAOTreasury - rate-limited treasury

- **Condition:** `totalWithdrawn` takes into account the total amount withdrawn from the treasury. The base budget is fixed upon initialization: `fixedBudget = balance + totalWithdrawn` when `initializeBudget()` is called; then `_totalBudget()` returns exactly `fixedBudget`.
- **Unlock:** `unlockedAmount()` simulates a geometric release schedule with a 1-day interval (`RELEASE_INTERVAL`) and a rate `RELEASE_RATE_BPS`.
- **Available for withdrawal:** `availableToWithdraw()` = unlocked − totalWithdrawn.
- **Withdrawal:** `withdraw(to, amount)` is accessible only to the owner and checks the limit; logs `TreasuryWithdrawal(to, amount, totalWithdrawn)`.
- **Preview:** `previewReleasedAt(timestamp)` shows unlocked/withdrawable for a given time.

DAOPool / DAOManager / ClueAccounts - weight, activity, accounting

- **DAOManager** sets boundaries and defaults (duration, quorum, execution window, limit of active participants, etc.).
- **DAOPool** maintains the stake and computes `effectiveWeight`, and also implements `rewardActivity`/`penalizeActivity`.
- **ClueAccounts** stores counters/accounting (proposal creation, votes, penalties, execution, etc.), which makes the behavior of participants measurable and verifiable.

7. Moderation and Arbitration Mechanics

Moderation and arbitration in CLUE are implemented as a formal on-chain process to ensure market quality: who votes, with what weight, when a case is escalated, how consensus is calculated and how rewards/penalties are distributed - everything is fixed by contracts, not by manual decisions.

7.1. Moderation

- **Moderation module:** `Moderation` + `ModerationRegistry` + `CommitteeCore`. Moderators form the primary quality signal of the market through risk flags.
- **Link to markets:** `MarketsLifecycle` and `MarketsAppeals` open cases, transition the market to the required statuses and transmit data to the moderation/arbitration layer.
- **Committee source:** `ModeratorsPool` (top candidates + weights) and `ModeratorsManager` (committee sizes, score/slashing steps, rounds parameters).
- **What does a moderator do:** participates in the moderation case, votes using flags, receives a reward for participation and a penalty for missed votes/deviations from the final outcome.

7.1.1. How to open a moderation case

a) **Market creation:** in `MarketsLifecycle.addMarket` the market is created with the status `Approved`.
b) **Dispute funding:** `marketFee` is transferred to `moderationRegistry` as a moderation reward pool.
c) **Creating a case:** `createCaseFromRegistry(...)` is called in `Moderation`, then the case is linked to the market (`setModerationCase`).
d) **Seed generation:** the base seed of the case is stored in the market (`setMarketBaseSeed`) and is used to deterministically build round committees.

7.1.2. Mechanics of case moderation

A moderation case in CLUE is a separate on-chain entity tied to a specific `marketId`. Its task is not “centralized censorship,” but a formal assessment of the market’s risk profile through a deterministic committee and the flag-based model.

a) **Case init:** when creating a market, a moderation case is opened, a reward pool is added to the case (from `marketFee`) and system parameters are recorded (creation time, seed, case type).
b) **Round state:** active round is calculated by `roundIndex = (now - createdAt) / roundSeconds`; this makes the transition between rounds predictable and testable.
c) **Committee formation:** From the roster subset of moderators, a round committee is formed with a limit on weights (`maxWeightBps`) so that no address dominates beyond the permissible limit.
d) **Voting:** each committee member submits one vote in the format `uint8 flags`. The vote data is written to the case registry and used in the final bit count.
e) **Finalize:** after all committee votes are cast, the case is finalized, the final `moderationFlags` are written to `MarketsRegistry`, and rewards/penalties are applied to participants according to contract logic.

Practical implication: moderation does not directly change the economics of trading, but adds an on-chain layer of market quality that transparently affects the reputation and incentives of moderators.

7.1.3. Formation of committee and weight

- **Active set snapshot:** When creating a case, a top subset of moderators is taken from the pool according to the `activeModeratorsLimit`, then a weighted order of candidates is formed.
- **Round mechanics:** for each round `roundIndex = (now - createdAt) / roundSeconds`.
- **Rotating committee:** The composition of the round is assembled from the participants who have already voted + candidates from the roster chunk.
- **Voting power normalization:** the final committee weights are normalized and capped by `maxWeightBps` so that one address does not monopolize the round.

7.1.4. Risk flags and the bit model

In moderation, the vote is `uint8 flags` (bit mask). The contract stores the bits, and the human-readable labels are specified by the frontend/indexer specification.

- **Flag limit:** `moderationFlagsCount` (default 5, maximum 8).
- **Current profile (5 flags):** `bit0` (label: adult), `bit1` (label: violence), `bit2` (label: restricted_sensitive), `bit3` (label: scam_risk), `bit4` (label: low_quality).
- **Masks:** `1, 2, 4, 8, 16` (in hex: `0x01, 0x02, 0x04, 0x08, 0x10`).

7.1.5. How is consensus on flags calculated?

- **Condition for finalizing the round:** the case is closed when `votes == committeeSize`.
- **For each bit:** The weighted support for the flag is summed, then the strict majority rule is applied:

`flag_i = 1 if weight_i * 2 > totalWeight`; otherwise `0`.

Important: the threshold is strict (>50%), not ≥50%.

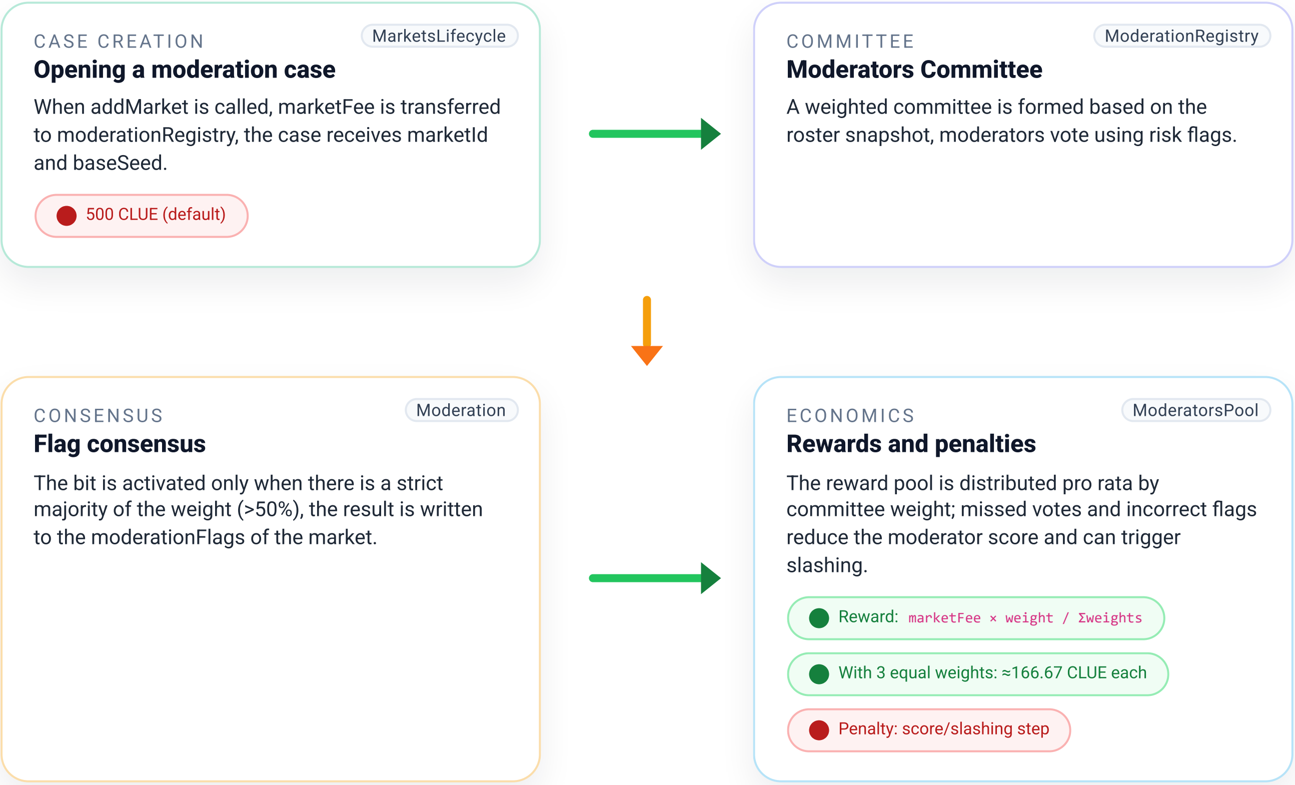
7.1.6. Rewards and penalties in moderation

- **Reward pool:** taken from funded `marketFee` and distributed among the final committee in proportion to the weights.
- **If there are no weights:** reward is returned to the creator address of the case.
- **Participation score:** the voter receives a reward based on activity; Missing votes in rounds are automatically penalized.
- **Slashing score:** after finalization, for each moderator, their bits are compared with the final ones. Matches give a recovery increment, discrepancies give a penalty increment (scaled by the proportion of correct/erroneous bits).

7.1.7. Moderator workflow diagram

Moderators Case Flow

Market add → Moderation case → Committee votes(flags) → Final flags → Reward/Penalty



CASE CREATION

MarketsLifecycle

Opening a moderation case

When addMarket is called, marketFee is transferred to moderationRegistry, the case receives marketId and baseSeed.

● 500 CLUE (default)

COMMITTEE

ModerationRegistry

Moderators Committee

A weighted committee is formed based on the roster snapshot, moderators vote using risk flags.

CONSENSUS

Moderation

Flag consensus

The bit is activated only when there is a strict majority of the weight (>50%), the result is written to the moderationFlags of the market.

ECONOMICS

ModeratorsPool

Rewards and penalties

The reward pool is distributed pro rata by committee weight; missed votes and incorrect flags reduce the moderator score and can trigger slashing.

● Reward: `marketFee × weight / Σweights`

● With 3 equal weights: ≈166.67 CLUE each

● Penalty: score/slashing step

7.2. Arbitration

Arbitrators get involved when the dispute over the outcome is not resolved at the stage of standard outcome resolution. Their task is to provide a final on-chain solution with measurable consensus and cryptoeconomic security.

7.2.1. When an appeal escalates to arbitration

- The market is in `AwaitingAppeals`, the outcome has already been submitted, the mode is only `Standard`.
- The appellant selects `desiredOutcome` (0 = cancel, 1..N = specific outcome), which must be different from the current outcome.
- One address - one appeal for the current epoch (`appealEpoch`).
- **Default escalation threshold:** `appealEscalationThresholdBps = 30_000` (that is **30%** in the `bpsDenom=100_000` scale).
- **What is opposing stake:** this is the amount of stake for all outcomes, *against* the selected `desiredOutcome`. In a binary market, this is literally the opposing side's stake; in multi-outcome, it is the sum of all alternative outcomes.
- **Escalation condition:** The appeal stake must cover at least 30% of opposing stake (by default), after which an arbitration case is created.
- **Appeal bond and deposit flow:** with `makeAppeal` an appeal fee deposit is made into the rewardPool; with `sendToResolve/reopenArbitration` a separate resolution bond is posted (minimum `minAppealBond`, default **100 CLUE**).

Formally: `appealedStake × 100000 ≥ stakeAgainst × thresholdBps`. In case of default `thresholdBps=30000` this is “30% of the opposing side's stake”.

7.2.2. Fee, stake and escalation threshold

- **Appeal fee:** is calculated as the maximum of the stake-based fee and floor component (manager parameters: `appealFeeBps`, `appealFeeFloorBps`).
- **Target outcome:** is fixed by the first appeal; subsequent appeals must maintain the same target.
- **Escalation:** occurs when the total appeal stake passes the threshold:

`appealedStake × 100000 ≥ stakeAgainst × appealEscalationThresholdBps`.

During escalation, an arbitration case is created, the reward pool of appeals is transferred to the bond pool, and the market is linked to the arbitration case id.

7.2.3. Auto-resolve, escalation and opening of an arbitration case

- **If the creator did not submit the outcome on time:** after `resolveAt + outcomeSubmissionGrace` any address can call `sendToResolve` by posting a bond.
- **If the arbitration consensus is insufficient:** the market remains in `AwaitingAppeals`, the appeal fee parameters are increased (`appealBondEscalationStepBps`), and the process can be repeated with a new arbitration round.

7.2.4. Mechanics of an arbitration case

An arbitration case starts when a dispute goes beyond a normal appeal: through the threshold of escalation, auto-resolve, or re-escalation after weak consensus. This is a separate cycle with its own `caseId`, committee and accumulation of consensus metrics.

a) **Case creation:** an arbitration case is created, the market is linked with `arbitrationCaseId`, and appeal funds are transferred to the case bond pool.
b) **Committee selection:** The registry builds a committee of arbitrators from a pool of moderators using seed/roster logic with voting power normalization and the `arbitrationMaxWeightBps` limit.
c) **Outcome voting:** arbitrators vote on `outcomeIndex` (including `0 = cancel`). The round closes when the votes of the entire committee have been collected.
d) **Round result:** the winning outcome is determined, the round `consensusBps` is fixed, and then lifecycle updates the aggregates `consensusSumBps` and `arbitrationRounds`.
e) **Finalization threshold:** if the average consensus reaches the threshold, the market is finalized; if not, the market remains in `AwaitingAppeals` and a new cycle is launched (new `appealEpoch` and potentially a new arbitration case).

7.2.4.1. Arbitrators' voting

- **Valid outcomes:** `outcomeIndex` in the range `0..N`, where `0` is market cancellation.
- **Finalization:** at `votes == committeeSize`.
- **Winning outcome:** outcome with maximum weight; in case of equality, a deterministic tie-break via the case seed is used.

7.2.5. Aggregate Consensus and Subsequent Arbitration Case

For each arbitration round, `consensusBps = (winningWeight × 100000) / sumWeights` in bps is calculated. Next, lifecycle calculates the average consensus across rounds:

`avgConsensus = consensusSumBps / arbitrationRounds`.

Final execution is allowed when `avgConsensus ≥ arbitrationFinalConsensusBps` (the threshold is set by `ModeratorsManager`).

If the threshold is not reached, the protocol does not “force” a controversial ending: the market remains in `AwaitingAppeals`, the next round of appeals accumulates, and at the new escalation threshold it opens a **repeated arbitration case** (new caseId in new appeal epoch).

a) The arbitrators' round is closed and the local `consensusBps` is recorded.
b) Lifecycle updates `consensusSumBps` and `arbitrationRounds`.
c) `avgConsensus` is calculated as `consensusSumBps / arbitrationRounds`.
d) If `avgConsensus` is below the threshold, the market returns to `AwaitingAppeals`, appeal parameters are escalated (increased).
e) After repeated escalation, a new arbitration case is opened and the cycle continues until the final consensus threshold is reached.

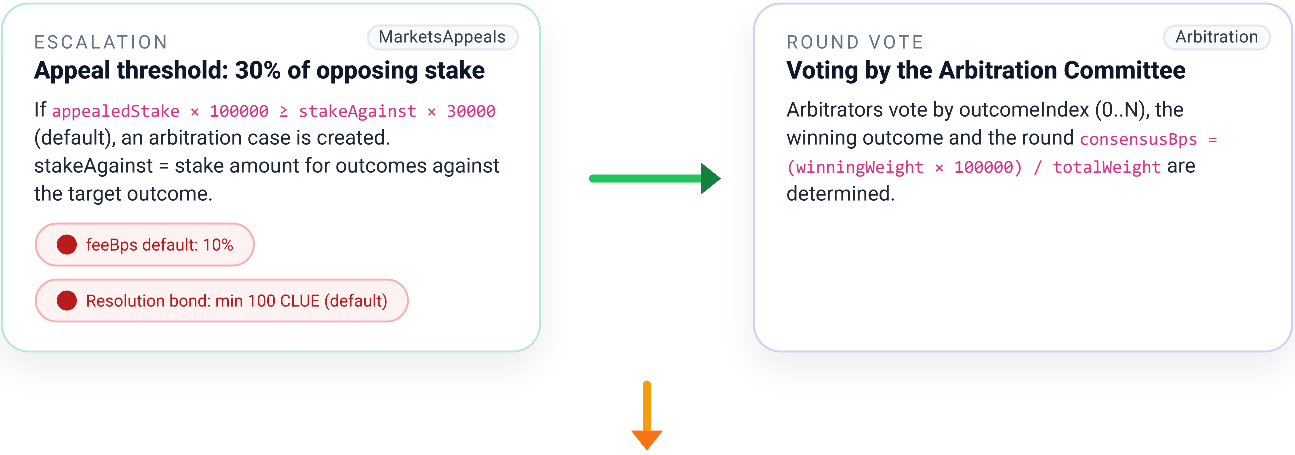
7.2.6. Reward, creator-share and penalties

- **Committee rewards:** are distributed proportionally to the weights of the final committee (if the case has a reward pool).
- **Creator fee reallocation:** if the appeal is upheld, the share of the creator fee can be redirected to the arbitrators.
- **Slashing score in arbitration:** arbitrators who voted for the final winning outcome receive a recovery increment; those voting against receive a penalty step.
- **Fallback mechanism:** If the transfer of a share is unsuccessful, pending withdrawals are recorded with the possibility of recovery through the user claim procedure.

7.2.7. Arbitration cycle diagram (with repeated case)

Arbitrators Consensus Loop

30% escalation vs opposing stake → Arbitration rounds → avgConsensus ≥ 70% ? → Finalize / New case



Appeal threshold: 30% of opposing stake

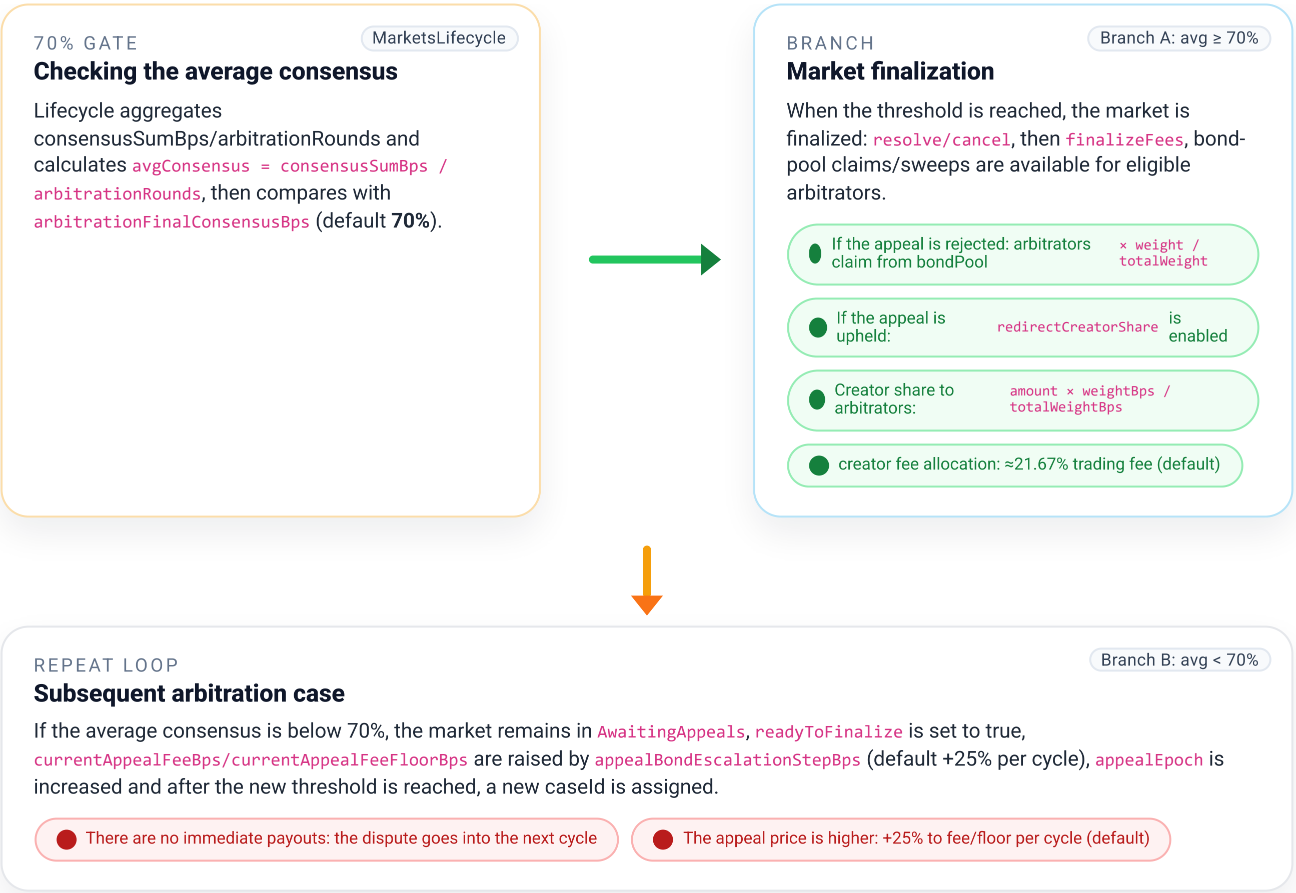
If `appealedStake × 100000 ≥ stakeAgainst × 30000` (default), an arbitration case is created. stakeAgainst = stake amount for outcomes against the target outcome.

● feeBps default: 10%

● Resolution bond: min 100 CLUE (default)

Voting by the Arbitration Committee

Arbitrators vote by outcomeIndex (0..N), the winning outcome and the round `consensusBps = (winningWeight × 100000) / totalWeight` are determined.



70% GATE

MarketsLifecycle

Checking the average consensus

Lifecycle aggregates consensusSumBps/arbitrationRounds and calculates `avgConsensus = consensusSumBps / arbitrationRounds`, then compares with `arbitrationFinalConsensusBps` (default **70%**).

BRANCH

Branch A: avg ≥ 70%

Market finalization

When the threshold is reached, the market is finalized: `resolve/cancel`, then `finalizeFees`, bond-pool claims/sweeps are available for eligible arbitrators.

- ● If the appeal is rejected: arbitrators claim from bondPool `× weight / totalWeight`
- ● If the appeal is upheld: `redirectCreatorShare` is enabled
- ● Creator share to arbitrators: `amount × weightBps / totalWeightBps`
- ● creator fee allocation: ≈21.67% trading fee (default)

REPEAT LOOP

Branch B: avg < 70%

Subsequent arbitration case

If the average consensus is below 70%, the market remains in `AwaitingAppeals`, `readyToFinalize` is set to true, `currentAppealFeeBps/currentAppealFeeFloorBps` are raised by `appealBondEscalationStepBps` (default +25% per cycle), `appealEpoch` is increased and after the new threshold is reached, a new caseId is assigned.

- ● There are no immediate payouts: the dispute goes into the next cycle
- ● The appeal price is higher: +25% to fee/floor per cycle (default)

Arbitration payout flows (deterministic)

- **What forms the arbitration pool:** appeal bonds (`rewardPool`) upon escalation go to `bondPool`; auto-resolve adds a separate caller bond.
- **Branch 1 - appeal upheld (targetOutcome matched the final outcome):** `redirectCreatorShare = true`, creator fee from AMM is sent to arbitration and divided by committee weights. The appeal pool is used for appellant refunds, and not for arbitrator payouts.
- **Branch 2 - appeal rejected (targetOutcome did not match):** the creator share remains with the market creator, and `bondPool` is distributed among eligible arbitrators in proportion to their weight; a claim window is in effect, after which the unclaimed balance can be swept into `feeTreasury` (claim window = `appealPeriod`, default 30 minutes).
- **Low consensus (<70% avg):** the market is not finalized, payouts remain locked, and a repeat cycle is launched with increased fee/floor appeal parameters and a new arbitration case.
- **Resolution-bond provider:** is refunded according to the resolution branch. If there is a temporary liquidity shortfall, a pending balance is recorded with a separate claim entry so that the amount is not lost.

7.2.8. Transparency and stability invariants

- **There is no manual “override” of outcomes:** status and outcome transitions are performed only through valid contract lifecycle branches.
- **Committees are determined by deterministic seed derivation:** the composition can be reproduced using on-chain data (baseSeed, roundIndex, active-set snapshot).
- **Voting power is not only stake:** The score and slashing metrics directly affect the future participation weight of the moderator/arbitrator.
- **Cryptoeconomic incentives:** rewards, bond pools, redirects and penalties form symmetrical incentives against passivity and dishonest decisions.

8. Market Mechanics

In CLUE, users create markets for events, trade outcomes and receive payouts based on the final result. The protocol does two things at once: ensures liquid trading (through AMM) and ensures the quality/fairness of markets (through moderation, appeals and arbitration).

Key principle - **execution of on-chain rules**. Smart contracts record market statuses, role rights, fees and final settlement. The interface and indexers are only an access layer for reading data: they have no privileges and cannot manage the funds.

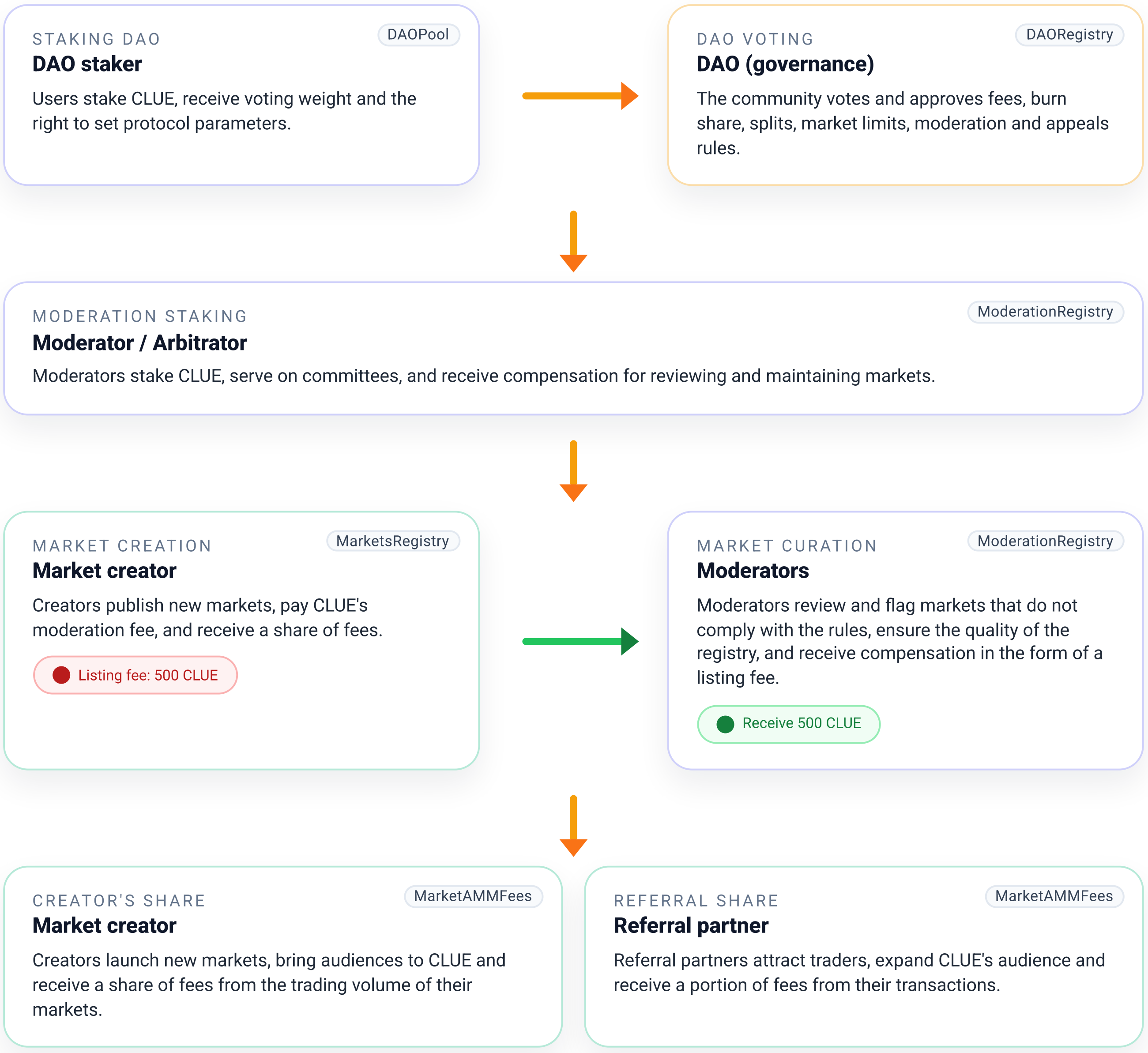
Parameters (fees, limits, appeal windows, module addresses, pause/protection modes) are controlled by the **DAO governance process and execution windows**. The team does not have a separate "admin key": critical changes are possible only through on-chain decisions.

8.1. How the market works

In CLUE, the market follows a clear path: creation → verification → trading → outcome submission → (if necessary) appeal/arbitration → final payout. At every step, participants have economic incentives, and the protocol has safeguards against abuse.

CLUE Ecosystem Diagram

Value flow: DAO → roles → growth → traders → AMM → fees → resolution → rewards → DAO feedback



STAKING DAO
DAOPool
DAO staker
Users stake CLUE, receive voting weight and the right to set protocol parameters.

DAO VOTING
DAORegistry
DAO (governance)
The community votes and approves fees, burn share, splits, market limits, moderation and appeals rules.

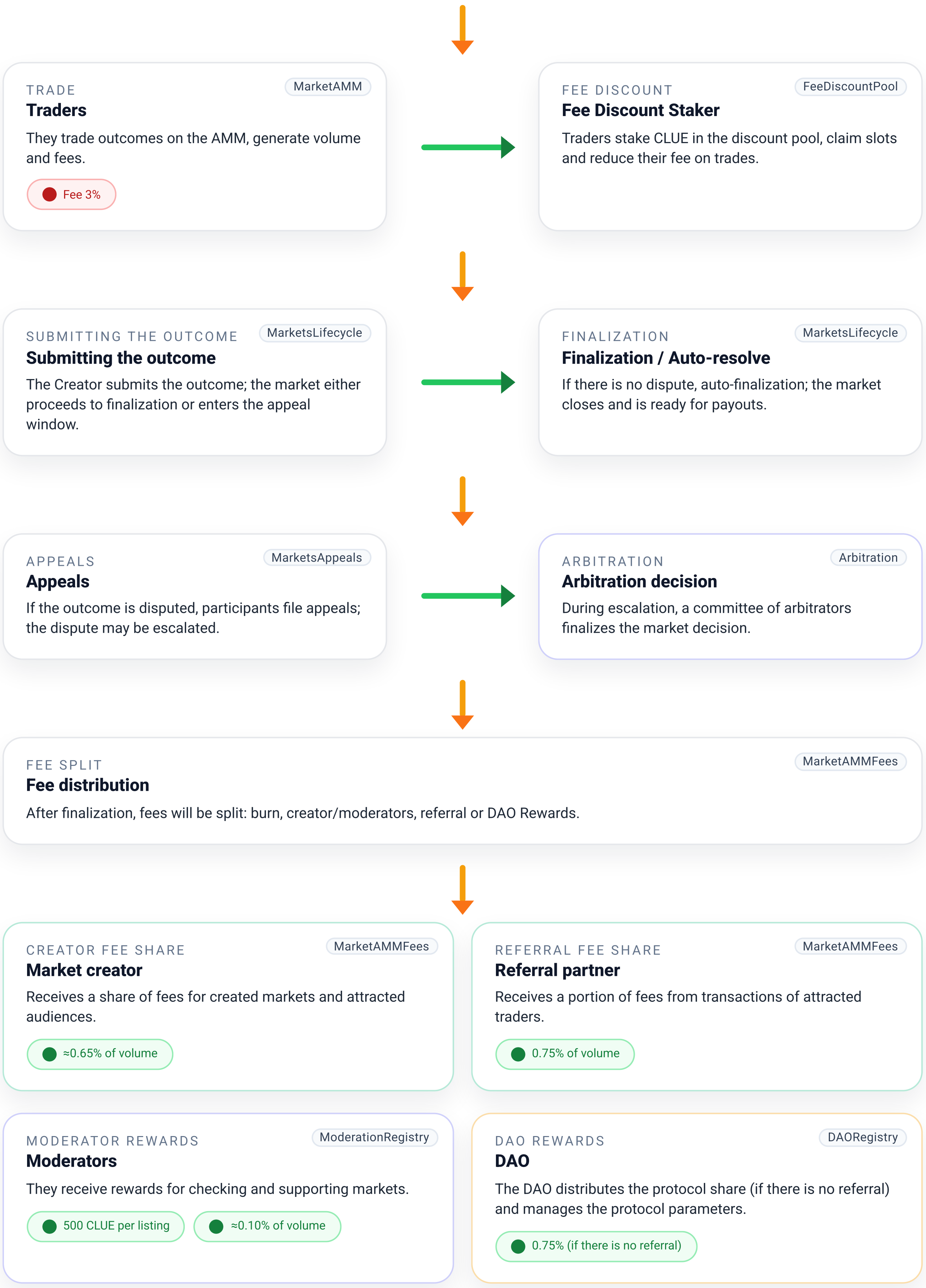
MODERATION STAKING
ModerationRegistry
Moderator / Arbitrator
Moderators stake CLUE, serve on committees, and receive compensation for reviewing and maintaining markets.

MARKET CREATION
MarketsRegistry
Market creator
Creators publish new markets, pay CLUE's moderation fee, and receive a share of fees.
● Listing fee: 500 CLUE

MARKET CURATION
ModerationRegistry
Moderators
Moderators review and flag markets that do not comply with the rules, ensure the quality of the registry, and receive compensation in the form of a listing fee.
● Receive 500 CLUE

CREATOR'S SHARE
MarketAMMFees
Market creator
Creators launch new markets, bring audiences to CLUE and receive a share of fees from the trading volume of their markets.

REFERRAL SHARE
MarketAMMFees
Referral partner
Referral partners attract traders, expand CLUE's audience and receive a portion of fees from their transactions.



TRADE

MarketAMM

Traders

They trade outcomes on the AMM, generate volume and fees.

● Fee 3%

FEE DISCOUNT

FeeDiscountPool

Fee Discount Staker

Traders stake CLUE in the discount pool, claim slots and reduce their fee on trades.

SUBMITTING THE OUTCOME

MarketsLifecycle

Submitting the outcome

The Creator submits the outcome; the market either proceeds to finalization or enters the appeal window.

FINALIZATION

MarketsLifecycle

Finalization / Auto-resolve

If there is no dispute, auto-finalization; the market closes and is ready for payouts.

APPEALS

MarketsAppeals

Appeals

If the outcome is disputed, participants file appeals; the dispute may be escalated.

ARBITRATION

Arbitration

Arbitration decision

During escalation, a committee of arbitrators finalizes the market decision.

FEE SPLIT

MarketAMMFees

Fee distribution

After finalization, fees will be split: burn, creator/moderators, referral or DAO Rewards.

CREATOR FEE SHARE

MarketAMMFees

Market creator

Receives a share of fees for created markets and attracted audiences.

● ≈0.65% of volume

REFERRAL FEE SHARE

MarketAMMFees

Referral partner

Receives a portion of fees from transactions of attracted traders.

● 0.75% of volume

MODERATOR REWARDS

ModerationRegistry

Moderators

They receive rewards for checking and supporting markets.

● 500 CLUE per listing

● ≈0.10% of volume

DAO REWARDS

DAORegistry

DAO

The DAO distributes the protocol share (if there is no referral) and manages the protocol parameters.

● 0.75% (if there is no referral)

BURN

AutoBurnTreasury

Burn

50% of the fee is sent to the burn treasury, which burns CLUE and increases token scarcity.

● 1.50% of volume

More markets → more trades → more fees → more burn and rewards → more referrals → more markets

8.1.1. Types of markets

All CLUE markets use a multi-outcome market model: an event is formalized as a set of mutually exclusive outcomes. The Yes/No format is a special case of the same model with two outcomes (Yes and No).

At the contract level, this means that the market is always described by a finite set of alternatives, and the outcome is fixed by choosing one of them (or canceling in the provided lifecycle scenarios). This approach makes the system machine-verifiable: each outcome has a unique index, trading liquidity is taken into account using the `q[]` vector, and payments after resolution are calculated deterministically. Important point: the protocol works with **discrete** outcomes, so complex "continuous" events (for example, the price of an asset as a number) must be broken down into understandable intervals/categories in advance.

- **Binary markets (2 outcomes)**: classic format *Yes/No* for questions with a clear binary outcome ("will happen / will not happen"). This is the easiest type to understand and usually the most liquid at the start.
- **Categorical markets (N outcomes)**: one market contains several mutually exclusive options (for example, "who will win", "which scenario will play out"). All outcomes are traded on the same AMM curve, maintaining consistent pricing between options.
- **Scalar markets via discretization**: Custom logic can define a "basket" of ranges (e.g. A/B/C/D), but for the protocol it is still a categorical market with a fixed number of mutually exclusive outcomes.
- **Model limitations**: the number of outcomes is limited by the protocol parameters (`maxOutcomesCount`), and each outcome must be defined in advance and unambiguously. This reduces ambiguity in resolution and simplifies the appeal process.

8.1.2. Market resolution types (Standard Resolution, Self Resolution)

The resolution type is set when creating a market using the `arbitrationMode` parameter in `addMarket(...)`: **Standard** (0) or **Self** (1). The current market registry uses only statuses **Approved**, **AwaitingAppeals**, **Finished**.

- **Standard Resolution (ArbitrationMode.Standard)**: after `submitOutcome` the market goes into **AwaitingAppeals**. During `appealPeriod`, participants can submit an appeal through `makeAppeal`. When the threshold is reached, the dispute is escalated to arbitration. If there are no appeals or escalations, the market is closed via `markFinished` followed by resolve/cancel via the AMM module and `finalizeFees`.
- **Resolution failsafe**: If the outcome is not submitted by `resolveAt + outcomeSubmissionGrace`, any address can call `sendToResolve` (by posting a bond), which triggers the arbitration process and prevents the market from getting stuck.
- **Self Resolution (ArbitrationMode.Self)**: only the creator can submit the outcome, and with `submitOutcome` the market is immediately transferred to **Finished**, with no appeal window and no arbitration; then AMM performs resolve/cancel and `finalizeFees`.
- **Key Difference**: Appeals and arbitration are only available to **Standard**; for **Self** these branches are disabled at the contract logic level.

8.1.3. Market moderation

In the current version of the protocol, moderation is implemented as **on-chain risk flagging** through cases in `ModerationRegistry`. This is important: moderation sets risk flags, but does not introduce a separate pre-trade status like Pending/Declined.

- **Case creation and funding**: when the market is published, `marketFee` is transferred to the moderation module, after which a moderation case is created with a committee of moderators and a reward pool.
- **Committee vote**: moderators vote with a bitmask of flags (`vote(..., flags)`). After a full set of committee votes, the case is finalized on-chain.
- **How flags are formed**: Weighted majority rule (>50% committee weight) is applied for each bit. The number of supported flags is specified by the `moderationFlagsCount` parameter (default 5, maximum 8).
- **Flag specification (current profile at `moderationFlagsCount = 5`)**:
  - `bit0` (label: adult) (mask `0x01`, value `1`)
  - `bit1` (label: violence) (mask `0x02`, value `2`)
  - `bit2` (label: restricted_sensitive) (mask `0x04`, value `4`)
  - `bit3` (label: scam_risk) (mask `0x08`, value `8`)
  - `bit4` (label: low_quality) (mask `0x10`, value `16`)

  At the contract level, only the `uint8` bitmask and majority bit logic are stored; semantic names are a catalog/indexer specification on top of on-chain bits.
- **What changes in the market**: in `MarketsRegistry` only the field `moderationFlags` is updated through `setModerationFlags`; the trading status remains within the framework of the main lifecycle model.
- **Economics and responsibility**: the reward of the case is distributed among moderators according to weights; missed rounds and incorrect decisions reduce score/slashing metrics, while correct ones restore them.

8.1.4. Price discovery (AMM)

In CLUE, the price is formed through the LMSR model. The quote is calculated on-chain from the current liquidity vector `q[]` and the liquidity parameter `b`, so buying/selling is always determined by the code and does not depend on the order book.

Basic formulas of the model

- **Cost function**: $C(\mathbf{q}, b) = b \cdot \ln\left(\sum_i e^{q_i/b}\right)$.
- **Marginal price of outcome i**: $p_i = \frac{e^{q_i/b}}{\sum_j e^{q_j/b}}$.
- **Buying delta Δ**: `cost = C(q + Δ_i, b) - C(q, b)`.
- **Selling delta Δ**: `rebate = C(q, b) - C(q - Δ_i, b)`.

Dynamic parameter `b`

In the contract `b` is not fixed “once and for all”. The dynamics used are: `b = max(baseB, alpha × liquidityProxy)`, where `alpha = (1 + overround) / (n × ln(n))`.

- **`baseB`** — lower threshold of curve sensitivity.
- **`liquidityProxy`** — a proxy for market depth (`_liquidityProxyShares()`) associated with the current treasury and payout parameters.
- **`bBoostShares`** sets a virtual liquidity floor so that the curve does not become too “sharp” on small volumes.

Trader experience

- **The more you buy one outcome, the higher the next price** of the same outcome (classic LMSR curve effect).
- **With increasing market depth** (via dynamic `b`) the curve becomes smoother, and slippage for the same trade size is usually lower.
- **The fee is calculated separately from the LMSR price.** For buying, the gross fee formula applies: `fee = cost × rate / (1 - rate)`, for sell: `fee = rebate × rate`. The base rate is set on-chain (3% by default) and is reduced by a personal discount from Fee Discount.

- **Quote protected by slippage limits**: The user specifies `maxCost`/`minOut`, and the contract additionally checks the exponential bound (`EXP_INPUT_LIMIT_WAD`) for the numerical stability of the model.

8.1.5. Settlement and payouts

In CLUE, payout fairness is ensured not by empty promises, but by a sequence of on-chain invariants: the market must first be finalized, then AMM performs a deterministic resolve (resolve/cancel), and only after that fees are distributed.

- **Finalization only in a valid state**: `finalizeFees` is allowed only when the market has already settled (`resolved` or `cancelled`) and has the final status `Finished`. This prevents early payouts during the dispute phase.
- **Redemption in a resolved market**: The user redeems only winning tokens. The amount is calculated as `payout = amountWinTokens × winPayoutPerToken`, where the coefficient is derived from the current treasury and the balance of winning shares (`treasury / q[winningOutcome]`). Thus, payments are distributed in proportion to the real winning-outcome balance.
- **Refunds in a canceled market**: the refund is based on the share of all the user's shares relative to the total volume: `payout = treasuryBefore × userShares / outstandingShares`. To eliminate rounding remainders, the last redeemer receives the remaining treasury balance in its entirety.
- **Double-claim protection**: before sending funds, positions are burned, internal stake/balance records are cleared, and the treasury is reduced by the calculated amount. This blocks repeated redemptions of the same position.
- **Disputes do not block redemptions**: with `redeem()` AMM tries to automatically claim the available `arbitration bond` and `appeal refund` for the user. If according to the rules they are not required, the call does not revert and the usual redeem is still executed.
- **Stablecoin settlement fallback**: if local AMM treasury is insufficient for a sell payout, the core module can unhedge through `MarketAMMFXTreasury.redeemTradingToken`; payout still remains bounded by slippage controls and on-chain balances.
- **Synthetic NO settlement consistency**: `MarketAMMNoAggregator` snapshots exposures and processes resolved/cancelled claims pro-rata, including remainder handling on the last claim, so NO-side payouts stay mechanically auditable.
- **Fair fee distribution**: the collected fee is divided on-chain into shares for burn/treasury, creator, moderation, referral/DAO. If the amount cannot be split evenly, the remainder is added to the burn allocation (nothing is "lost"). If the appeal is upheld, the creator's share is redirected to arbitration (`redirectCreatorShare`).
- **Pull payment pattern**: if the transfer to any recipient does not go through, the amount is not burned, but remains in the pending pool of the corresponding category and can be sent again through retry functions. This reduces the operational risk of stuck distributions.
- **Burn mechanism**: a share of fees is sent to the burn mechanism, forming deflationary pressure on CLUE in the parameters approved by the DAO.

8.1.6. Referral model and growth

The CLUE referral model works directly at the level of AMM trading fees: in each transaction you can pass the referral address `referral`, and then a share of the fee is accumulated for this partner. If `referral` is not specified, the corresponding share goes to DAO rewards. The model parameters are configured by the DAO, but current contract defaults use the values below.

- **Basic figures (default)**: trading fee 3% (`ammTradeFeeBps` = 3000 bps), referral share - 25% from the collected fee (`_ammFeeShareReferralBps` = 25000 of 100000 bps). Equivalent to 0.75% from trading volume at a fee of 3%.
- **Calculation formula**: `referralTokens = feeTokens × referralShareBps / 100000`. Important: reward is calculated from **the actually paid fee**, and not from the nominal volume of the transaction.
- **Discount interaction**: if a trader has a fee-discount, their effective fee is lower, which means that the absolute referral payout is also lower, but the referral share (as a percentage of the collected fee) remains unchanged.
- **Example 1 (without discount)**: transaction volume 10 000 CLUE, fee 3% = 300 CLUE, referral-share 25% ⇒ partner receives 75 CLUE.

- **Example 2 (base 3% and discount 50%)**: volume 10 000 CLUE, basic fee 3%, discount 50% ⇒ effective fee 1.5% = 150 CLUE, referral-share 25% ⇒ partner receives 37.5 CLUE. The trader's saving relative to the base fee is 150 CLUE, and the partner's reward, as before, is calculated from the fee actually paid.
- **When and how to receive the reward**: referral amounts are accumulated in the fee module of a specific market and become available to the referrer after fee finalization (`claimReferralReward` at `feesFinalized`).
- **If there is no referral**: the referral part is not lost, but goes into the DAO rewards pool, which maintains the economic balance of the split and maintains the overall incentive pool of the ecosystem.
- **Key growth metrics**: trading volume, number of unique traders, retention, burn share and quality of the market catalog (appeal rate and share of correctly finalized markets).

8.2. Technical Specification

The technical specification specifies the design of the protocol: layers of the system, key invariants, security prerequisites, basic data flows and component connections that ensure the implementation of the rules in practice. All formulations below are based on current smart contracts from the repository (Markets/AMM/Appeals/Moderation/DAO/Discounts/Accounts) and do not assume the presence of external privileged oracles: the outcome is submitted by the market creator, disputed through appeals and, if necessary, escalated to arbitration.

In this section, the protocol is treated as a connected set of on-chain state machines, where each critical action is validated by contracts against roles, statuses, deadlines, and module configuration. In practical terms, this means the `creation → moderation → trading → outcome submission → appeals/arbitration → finalization → payouts` path cannot skip required steps without `revert`. The trust boundary is strict: off-chain components (indexers, UI) improve read performance and UX, but they cannot change market state, move funds, or override outcomes.

Architecturally, the protocol is divided into: market life cycle orchestration (`Markets`/`MarketsLifecycle`/`MarketsRegistry`), trading mathematics and AMM calculations (`MarketAMM`, `MarketAMMMath`, `MarketAMMFees`), dispute and quality layer (`Moderation`, `MarketsAppeals`, `ModerationRegistry`, `Arbitration`), as well as governance/parameters (`ClueManager`, module managers, the DAO and timelock). This decomposition reduces coupling, simplifies auditing, and allows the DAO to change parameters and module addresses without manual intervention in user positions.

- **Execution determinism**: prices, fees, splits and payments are calculated on-chain according to fixed formulas and bps parameters of managers.
- **Explicit Lifecycle Invariants**: trading is allowed only in valid statuses; fees are finalized only after the market has settled; appeals are limited by time windows and escalation thresholds.
- **Fallback behavior**: for unsuccessful payments, pending pools and retry mechanisms are used so that funds are not "lost" due to operational failures of the recipient.
- **Governance without hidden privileges**: critical changes are made through DAO governance and on-chain calls `set*` in managers, with a transparent history of events.

8.2.1. Contract interactions (core mechanics)

- **Market creation**: `Markets.addMarket` (delegated to `MarketsLifecycle.addMarket`) validates `closeAt/resolveAt/outcomesCount`, transfers `marketFee` to `moderationRegistry`, creates an entry in `MarketsRegistry` and **immediately** sets status `Approved` (there is no `Pending/Declined` in the current enum), after which it deploys AMM via `_deployApprovedMarketAMM`. Then a moderation case is created via `IModerationCaseFactory.createCaseFromRegistry` and linked to the market.
- **Moderation and flags**: in the current version `Moderation/ModerationRegistry` finalize `moderationFlags` (bitmask) and write them to `MarketsRegistry.setModerationFlags`. That is, moderation affects the risk profile of the market, but does not switch it to separate pre-trade statuses.
- **Trading**: allowed as long as the market is `Approved`, not `resolved/cancelled` and the current time is less than `closeAt`. `buyDelta/sellDelta` use LMSR (`MarketAMMMath`), update `q[]`, `totalRealShares`, stake accounting and participant registration in `MarketAMMRegistry`. Checks include `EXP_INPUT_LIMIT_WAD`, deadline/slippage, same-block execution guard and correct configuration of fee/resolution modules.
- **FX-enabled trade path**: in FX mode `buyDeltaWithFx/sellDeltaWithFx` are used; buy-side net flow can be hedged via `MarketAMMFXTreasury.hedgeToStable`, and sell-side can unhedge via `redeemTradingToken` when local treasury is short.
- **Synthetic NO path**: `MarketAMMNoAggregator` composes NO from YES legs across all outcomes except excluded one, with strict coverage, dedupe and slippage guards; in core trading, `manager.ammNoAggregator()` is the only same-block execution guard exception.

- **Outcome submission and resolution**: submitOutcome is available after resolveAt. In Standard mode, only the creator submits in the grace window. After grace, any market participant can also submit the outcome. The status is changed to AwaitingAppeals. In the Self mode, only the creator submits the outcome, and the market immediately transitions to Finished (without an appeal window).
- **Appeals/escalation**: makeAppeal works only for Standard with status AwaitingAppeals and within appealPeriod. The first appeal records targetOutcome, and the escalation is triggered by the stake threshold (via appealEscalationThresholdBps) and creates an arbitration case. If the outcome is not submitted before resolveAt + outcomeSubmissionGrace, anyone can call sendToResolve by posting a bond.
- **Completion and Settlement**: the final settlement is executed via resolveFromRegistry/cancelFromRegistry. In a resolved market, redemption burns winning-outcome tokens, and winPayoutPerToken is calculated from the current treasury and the balance of winning-outcome shares. In a canceled market, payout is calculated according to the proportion treasury × userShares / outstandingShares. The last redeemer receives the remainder of the treasury. Appeal refunds are available via claimAppealRefund only with refundMode.
- **Fees**: MarketAMMFees.recordFee accumulates fee, divides it into shares from MarketsManager.ammFeeDistributionBps (buyback/creator/moderation/referral) and sends the referral share either to the referral or to the DAO rewards pool if there is zero referral. finalizeFees is allowed only after settlement and status Finished. With redirectCreatorShare the creator's share goes to the arbitration module.
- **Discounts**: FeeDiscountPool + FeeDiscountRegistry.currentDiscountBps reduce the trader's effective fee; the proportions of the split do not change. The savings are recorded in ClueAccounts as RewardKind.FeeDiscount.
- **Referrals**: in the current core flow, referral amounts are stored in MarketAMM (referralRewards) and are claimed via claimReferralReward(). If there is no referral, the corresponding share goes to feeDaoRewardsTokens and then to the DAO distributor.
- **Security**: reentrancy guards in critical paths, module connectivity checks via manager/registry, strict roles for admin calls, ClueERC20 with pause/unpause owner (DAO). The score/weight of moderators is automatically reduced (activityBps/slashingBps), and direct token slashing is done with a separate DAO call slashStake(...).

8.2.2. System layers

- **On-chain**: routing and config via ClueManager + managers; market and lifecycle via Markets/MarketsLifecycle/MarketsAppeals/MarketsRegistry; AMM clones MarketAMM with Core/Fee/Resolution modules; optional FX lane via MarketAMMFXTreasury + ammFxPoolManager; synthetic NO lane via MarketAMMNoAggregator; moderation/arbitration via Moderation/ModerationRegistry/Arbitration; DAO governance via DAORegistry/DAOPool/DAOManager (optional DAOExecutor, treasury DAOTreasury); discounts via FeeDiscountPool/FeeDiscountRegistry; user account via ClueAccounts.
- **Off-chain**: indexers (event aggregation, cache for UI) and frontend (SPA). These components are not privileged: they speed up data access and improve UX, but cannot change on-chain state or manage funds.
- **Resolution & disputes**: in the Standard mode, the outcome in the grace window is submitted by the creator, after grace - also by the market participant; in the Self mode, only the creator submits the outcome and the market is immediately finalized. Dissenters initiate appeals via MarketsAppeals. When the stake threshold is reached, an arbitration case is created, and the final decision is recorded on-chain via ModerationRegistry/Arbitration.
- **Governance lane**: governance proposals and execution go through DAORegistry (voting deadline, resolve-grace, execution-window from DAOManager). Changes to parameters/addresses are made through on-chain calls to ClueManager and managers and can be executed directly from the DAO registry or through the optional DAOExecutor. There are no off-chain privileged control paths in the core logic.

8.2.3. FX hedging module (AMM ↔ stable)

- **Purpose:** isolate part of AMM trading-token inflow into a stable lane and restore it on demand for payouts, reducing treasury shock during high volatility.
- **Execution module:** MarketAMMFXTreasury performs hedge/unhedge through an external Uniswap-v4-compatible PoolManager (unlock/swap/sync/settle/take flow).
- **Safety gates:** registry-bound AMM validation, pool defaults from manager, non-zero pool params, and slippage controls via minOut/maxIn.
- **Upgrade continuity:** on AMM replacement, MarketsLifecycle.updateMarketAMM migrates stable balance via migrateStableBalance(oldAmm,newAmm,marketId).

8.2.4. Synthetic NO lane (MarketAMMNoAggregator)

- **Purpose:** provide synthetic NO exposure for multi-outcome markets by buying/selling the full YES basket except the excluded outcome.
- **Entry constraints:** full coverage required (`legs = N-1`), duplicate outcomes are rejected, and per-leg deltas must be equal; partial or malformed baskets revert.
- **Exit logic:** unwind is pro-rata by recorded leg exposure, with per-leg min-out allocation and final aggregate slippage check (`payoutTokens >= minPayoutTokens`).
- **Settlement:** resolved/cancelled redemption is snapshot-based, with deterministic pro-rata accounting and remainder-safe final claim handling.

8.2.5. Slippage, execution and abuse guards

- **User-side price guards:** buy path is bounded by `maxCostTokens`, sell path by `minOutTokens`, and both are additionally bounded by `deadline`.
- **Math/overflow guards:** AMM enforces LMSR exponent bound `EXP_INPUT_LIMIT_WAD` and liquidity checks to prevent unstable deltas.
- **Execution-pattern guard:** same-block repeat trading is blocked by default; only configured `ammNoAggregator` has exception scope for bundled synthetic-NO operations.
- **FX guardrail:** unhedge calls are constrained by `minOut`/`maxIn` and treasury shortfall checks before payout, preventing silent underfill.

8.2.6. Architectural diagram (contracts and connections)

Key protocol domains (high-level):

- **Markets and AMM**: creating a market via `MarketsLifecycle`, recording in `MarketsRegistry`, deploying an AMM clone and trading; accounting of outcome positions is maintained in the AMM state and is used for settlement/redemption.
- **Moderation and disputes**: moderation sets risk flags (`moderationFlags`), then the appeal process and (in case of escalation) arbitration operate, after which the final on-chain decision is recorded.
- **Accounting and parameters**: `ClueManager` and address/parameter managers, discounts and accounting of operations and fees.
- **DAO**: voting and execution via `DAORegistry` (and optionally `DAOExecutor`), changing parameters/addresses, treasury and distribution management.

Contract connections:

- **Registry, configuration, and parameters**
  - `ClueManager` → stores module addresses: `marketsContract`, `marketsLifecycleContract`, `marketsAppealsContract`, `marketsRegistry`, `ammRegistry`, `moderationContract`, `moderationRegistry`, `arbitrationContract`, `daoRegistry`, `daoPool`, `daoExecutor`, `feeDiscountRegistry`, `clueAccounts`, `ammContract`, `ammFeesContract`, `ammResolutionModule`, `ammTradingToken`, `ammFxTreasury`, `buybackTreasury`, `feeTreasury`.
  - `ClueManager` ⇄ parameter managers: `MarketsManager`, `ModeratorsManager`, `DAOManager`, `FeeDiscountManager`.
- **Markets and AMM**
  - `Markets` (router) → `delegatecall` to `MarketsLifecycle` and `MarketsAppeals`.
  - `MarketsLifecycle.addMarket` → `MarketsRegistry.createMarket`; `marketFee` (CLUE) is transferred to `ModerationRegistry`; a moderation case is created via `Moderation.createCaseFromRegistry`.
  - `MarketsLifecycle` → clones `MarketAMM` (template from `MarketsManager.ammContract()`) and links via `MarketAMMRegistry.setMarketAMM`.
  - `MarketAMM.initialize` → validates the `market ↔ registry ↔ manager` bundle, loads/wires `core/fees/resolution` modules and the trading token from the manager layer.
  - `OutcomeToken` (contract in the project) - minimal ERC1155-like outcome balance module with `lifecycle/controller` access; in the current core flow, AMM uses internal `outcomeBalances`, so `OutcomeToken` is not part of the main settlement path (see 6.2).
  - `MarketAMMCoreModule` ↔ `MarketAMMMath`: buy/sell, update `q[]`, stake and positions; participant registration via `MarketAMMRegistry.registerParticipant`.

  - `MarketAMMNoAggregator` - aggregator of synthetic NO position for multi-outcome markets; in core trading, AMM is used as a permitted exception for the same-block execution guard (address `manager.ammNoAggregator()`).
  - `MarketAMMQuoter` ↔ `MarketAMM` (via `IMarketAMMState/IMarketAMMFeesProbe`) provides view quotes for buy/sell/price; complex quote-by-spend/quote-by-payout calculations are delegated to `MarketAMMQuoteHelper`.
  - `MarketAMMCoreModule` → `FeeDiscountRegistry.currentDiscountBps` and `recordFeeSavings` (fee savings counter).
  - `MarketAMMFees.recordFee` → split by `buyback / creator / moderators / referral|dao`; `finalizeFees` is executed only after `Finished`.
  - `MarketAMMFees` → payouts: `buybackTreasury` (fallback in `feeTreasury`), `moderationContract`, creator/submitter, or `Arbitration.distributeCreatorShare` with `redirectCreatorShare`. The DAO share is forwarded via `DAORewardsDistributor.forwardReward`.
  - `MarketAMMResolutionModule` ← calls from `Markets`/`MarketsLifecycle` and `Arbitration` (resolve/cancel/redeem).
  - `MarketAMMFXTreasury` ↔ `MarketAMMCoreModule/resolutionModule`: hedge trading-token → stable and reverse unhedge; migration of stable balance at `MarketsLifecycle.updateMarketAMM`.
- **Moderation, appeals, arbitration**
  - `ModeratorsPool` - staking/weights/score/slashing for moderators and arbitrators; `ModerationRegistry` is used for selecting committees, as well as `Moderation`/`Arbitration` for reward/penalty paths.
  - `Moderation` and `Arbitration` work through the common `ModerationRegistry` (cases, committees, weights, vote-records).
  - `ModerationRegistry` deploys the read-only module `ModerationRegistryView` (via `ModerationRegistryViewProxy`) for deterministic view operations on cases/committees without changing state.
  - `Moderation.vote*` → `ModerationRegistry.voteFlagsFromModule`; finalizing the case puts `moderationFlags` into `MarketsRegistry` through `Markets.setModerationFlags`.
  - `MarketsAppeals.makeAppeal` → checking stake by AMM positions, lock via `MarketAMM.lockAppealStake`, accumulation `rewardPool`/`bondPool`.
  - Escalation of appeals → `Arbitration.createArbitrationCaseFromRegistry` through the case factory; the case is linked to `MarketsRegistry.setArbitrationCase`.
  - `ModerationRegistry` after the final of the arbitration case calls `Arbitration.onArbitrationFinalized` and then `Markets.resolveByArbitration`.
  - `MarketsLifecycle.resolveByArbitration` → records the outcome in the registry, completes the AMM, distributes the appeal bond pool and enables claim/sweep for arbitrators.
- **Accounting and discounts**
  - `ClueAccounts` - a single ledger of counters and rewards; writers are limited to roles from the manager layer (DAO/Markets/Moderation/Arbitration/FeeDiscount).
  - `FeeDiscountPool` → `FeeDiscountRegistry.notifyStakeChanged` when stake changes.
  - `FeeDiscountRegistry` calculates leaderboard, assigns slots/seats and writes state/indexes back to `ClueAccounts`.
  - `FeeDiscountRegistry.recordFeeSavings` accepts calls only from a valid AMM (`ammRegistry.marketAMM(marketId)`) and writes a reward `RewardKind.FeeDiscount` to `ClueAccounts`.
- **DAO and treasury**
  - `ClueERC20` — basic token-layer of the protocol; its address is stored in `ClueManager.clueToken()` and is used by markets/pools/treasuries in fee/reward/staking flows. The burn path is implemented via `AutoBurnTreasury`.
  - `DAORegistry` ⇄ `DAOManager`/`DAOPool`: creates proposals, calculates quorum/snapshot, finalizes and executes `target.call(actionCalldata)`.
  - `DAOExecutor` - optional execution module; authorized via `daoRegistry` (or owner if registry is not specified).
  - `DAORewardsDistributor.forwardReward` → transfers tokens to `DAORegistry` and calls `distributeDaoRewards`.
  - `DAORegistry.distributeDaoRewards` → distribution by `daoPool.eligibleTop/effectiveWeight`, then participants claim via `claimDaoReward`.

- `DAOTreasury` and `LiquidityPool`/`LiquidityPoolTreasury` are separate vesting treasury contracts with `withdraw` within unlocked limits.
- `AutoBurnTreasury` implements `IBuybackBurner`: when receiving CLUE from the fee module, tokens are burned automatically.
- **Top level coverage:** the diagram reflects all contracts from `contracts/*.sol` (33/33), including read-only service and aggregator modules (`MarketAMMQuoter`, `MarketAMMQuoteHelper`, `MarketAMMNoAggregator`, `ModerationRegistryView`, `OutcomeToken`).

8.2.7. Fee flow

Below is the actual on-chain flow of fees for current contracts `MarketAMMCoreModule` + `MarketAMMFees`.

Fee flow (in steps):

1. The transaction is routed via `buyDelta/sellDelta` to `MarketAMMCoreModule`; the fee is calculated in wad (`feeWad`) separately from the LMSR price.
2. `_collectFee` delegates to `MarketAMMFees.recordFee(feeWad, referral)` for both buy and sell paths.
3. `recordFee` converts wad → tokens (`_accrueFeeTokens`, the remainder is stored in `feeRemainderWad`) and distributes them into fee allocations `feeBuybackTokens`/`feeCreatorTokens`/`feeModeratorsTokens` + referral/DAO branch.
4. Until the market is finalized, no transfers are made: the values are only accumulated in fee allocations and in `referralRewards[referral]`.
5. After settlement and status `Finished`, `finalizeFees` is called: sending to `buybackTreasury` (fallback: `feeTreasury`), `moderationContract`, creator / arbitration and DAO distributor. Referral rewards are claimed separately via `claimReferralReward`.

- **Fee rate and discount**: **`f_base = ammTradeFeeBps / bpsDenom`**, **`f_eff = f_base × (bpsDenom - discountBps) / bpsDenom`** (in the code the values are capped by `bpsDenom`). Default now: 3% (`3000 / 100000`).

  The discount only reduces the fee, but does not change the LMSR mathematics and bps split proportions.

- **Fee formulas in the core module**: **`buy: fee = cost × f_eff / (1 - f_eff)`**, **`sell: fee = rebate × f_eff`**. For buy, the actual input token is **`ceil(cost + fee)`** (`_fromWadUp`), for sell payout to the trader — **`floor(rebate - fee)`** (`_fromWad`).

- **Fee split** is calculated from converted `feeTokens` according to `MarketsManager.ammFeeDistributionBps()`. Current default: 50% buyback 21.67% creator 3.33% moderators 25% referral/DAO.

  Due to integer division, the remainder of the split is added to the burn allocation; this behavior is hardcoded into `_splitFeeTokens`.

- **Referral/DAO branch**: if in `recordFee referral != 0`, the share goes to `referralRewards[referral]` (claimable only after `feesFinalized` via `claimReferralReward`). If `referral == 0`, the share goes to `feeDaoRewardsTokens` and is forwarded through `DAORewardsDistributor.forwardReward` at finalization.

  Important: the sell path can include `referral`; if it is `0`, the referral share of sell fees ends up in the DAO branch.

- **Creator branch and redirect**: with the normal finalization path, the creator fee allocation goes to `creatorReceiver` (creator or submitter, if the lifecycle contract specified this). When `redirectCreatorShare = true` the creator fee allocation goes to `Arbitration.distributeCreatorShare(caseId,...)`. Important: the same fee allocation may also contain an outcome bond written via `recordOutcomeBond`. If transfers fail, the amounts are not lost: they remain as pending payout balances and are retried via `retryCreatorPayout/retryArbitrationPayout/retryModeratorsPayout/retryBuybackPayout/retryDaoRewardsPayout`.

- **Volume equivalents at 3%**: if we take `V` as the gross basis of the transaction (buy: `pull`, sell: `rebate`), then `fee = 0.03 × V`, and the shares are 1.50% buyback ≈0.65% creator ≈0.10% moderators 0.75% referral/DAO.

  For buy, if you count from `cost` (before fee), the effective share will be `3% / (1 - 3%) ≈ 3.0928%`.

8.2.8. Security and invariants

- **No uncollateralized minting**: payments are limited to on-chain balances: redeem is limited to the pool treasury (`treasury`, with verification `TreasuryShortfall`), and fee payments are limited to accumulated fee allocations and the actual balance of the token; outcome tokens are minted into `buyDelta` only after the actual pull of the trading token.
- **Market trading gateway**: you can only trade when `status == Approved`, the market is not `resolved/cancelled`, the current time is less than `closeAt`, and the AMM in the registry is tied to this market. Any deviation gives `TradingClosed`/`revert`.
- **AMM invariants**: LMSR cost function, exponent constraint (`EXP_INPUT_LIMIT_WAD`) and bounds/overflow checks in math branches. This protects against numerical instability during extreme deltas.
- **FX hedging invariants**: hedge/unhedge operations are bounded by AMM registry binding and slippage parameters; if unhedge cannot satisfy payout bounds, execution reverts instead of creating hidden debt.
- **Fee invariants**: for buy the fee is calculated according to the gross fee formula `fee = cost × rate / (1 − rate)`, for sell - `fee = rebate × rate`; the effective rate is reduced by the discount (`discountBps`), but does not change the split proportions. The base rate and shares are further limited by the manager's bps validation.
- **Invariants of market finalization**: settle is executed only once via `resolveFromRegistry`/`cancelFromRegistry`. The winning outcome index is validated, and in the case of zero winning-outcome liquidity, the market correctly transitions to the cancellation flow.
- **Fee finalization invariants**: `finalizeFees` is allowed only after settlement and only with the final status `Finished`; this eliminates the premature withdrawal of fees during the dispute phase.
- **Position accounting invariants**: buy/sell synchronously updates `q[]`, `totalRealShares`, stake metrics and balance of outcome positions; redeem/redeemCancelled burn positions before payment, preventing double-claiming of the same share.
- **Synthetic NO invariants**: NO aggregator keeps per-leg exposure, validates full-basket coverage, and settles only through deterministic snapshot claims, preventing phantom NO balances.
- **Reentrancy + pause**: protection modifiers in trade/lifecycle/appeals and guard critical paths (primarily AMM, Markets, ModerationRegistry); CLUE token - `Pausable`, transfer is blocked when paused.
- **Roles and module connectivity**: sensitive operations are available only to authorized services/modules (markets/moderation/arbitration/registry/DAO-lane), and routers check the module's connection with the current manager (`ModuleManagerMismatch`) to exclude calls to “foreign” modules.
- **Deadlines and rights**: trade only with status **Approved** and up to `closeAt`; the outcome is submitted by the **creator** in the `outcomeSubmissionGrace` window, and after the window (in Standard mode) - also a market participant; appeals - within `appealPeriod`; `markFinished`/`finalizeFees` are executed only with valid module connections and valid final states.
- **Slashing/penalties**: automatic penalty - reduction of the payout ratio (`slashingBps`) for moderators/arbitrators; direct slashing of tokens is only possible through DAO governance (`DAORegistry`/`DAOExecutor`).

8.2.9. Edge cases and fallback safeguards

- **Market creation**: strict checks `closeAt/resolveAt`, number of outcomes, arbitration mode and mandatory modules; if violated - `revert`, the market does not appear in the registry.
- **Parameter bounds during creation**: `closeAt > now + minExpiryDuration`, `resolveAt > closeAt`, `resolveAt ≥ closeAt + minExpiryToEventGap`, maximum market duration limitation and `2 ≤ outcomesCount ≤ maxOutcomesCount` range are checked.
- **Deploying AMM and modules**: initialization fails if there is any inconsistency (incorrect registry/manager, missing template, unconfirmed `core/fees/resolution` module, incorrect market→AMM binding).
- **Trading and Liquidity**: `maxCost`/`minOut` limit slippage, `EXP_INPUT_LIMIT_WAD` blocks impossible values, the absence of a fee module or AMM connection gives `revert`.
- **Protection against market-manipulation patterns**: `deadline`, zero/incorrect delta, attempt to sell over balance, as well as same-block execution guard are checked (repeated trade in the same block for the same address is blocked, except for an authorized aggregator).
- **Overdue outcome submission**: if the outcome is not submitted, in Standard mode after `resolveAt + outcomeSubmissionGrace` anyone can call `sendToResolve` by posting a bond (or run via `markFinished` auto-escalation path) so that the market does not freeze.

- **Edge-case appeals**: targetOutcome is fixed by the first appeal of the epoch; one participant cannot make two appeals in the same epoch, and an appeal for a different outcome receives AppealOutcomeMismatch.
- **Appeals and refunds**: targetOutcome is fixed by the first appeal; refunding of deposits is possible only with active refundMode and non-empty rewardPool via claimAppealRefund.
- **Low Arbitration Consensus**: if the final consensus is below arbitrationFinalConsensusBps, the market does not finalize and instead enters another appeal cycle with escalation of appeal-bond parameters (appealBondEscalationStepBps).
- **Claim window for arbitration bond**: eligible arbitrators/moderators claim rewards within the window, after which the unclaimed remainder can be swept into feeTreasury; this finalizes distribution of any unclaimed rewards.
- **Pull payment pattern for fees**: if the transfer to the recipient is unsuccessful, the amount remains in the pending payout balance and is available for resending through the retry* functions (retryCreatorPayout, retryModeratorsPayout, retryDaoRewardsPayout, etc.).
- **Upgrade safety**: module addresses and parameters are changed only by the owner of the managers (DAO through governance calls to DAORegistry.finalize within the execution window). There are no hidden admin keys or backdoors to bypass the manager.
- **Status registry and risk catalog**: the market only has Approved → AwaitingAppeals → Finished, and moderation influences through moderationFlags; this eliminates “non-transparent” off-chain statuses and makes edge-cases verifiable against a single source of truth: smart contracts.

8.2.10. State machines

Market life cycle:

- **Creation (instant transition to Approved)**: The current MarketsRegistry.Status does not have Draft/Pending/Declined. After addMarket the market is immediately recorded as Approved, AMM is deployed and a moderation case is created; moderation is further influenced by risk flags (moderationFlags), and not by trading status.
- **Trading phase (derived-state in Approved)**: buy/sell are allowed as long as the following conditions are simultaneously met: status == Approved, block.timestamp < closeAt, !resolved, !cancelled and the correct market→AMM binding in the registry.
- **Submitting the outcome - Self branch**: creator only, after resolveAt; if submitOutcome is successful, the market immediately transitions to Finished, then resolve/cancel is executed via the AMM module and finalizeFees (without appeal window).
- **Submission of outcome - Standard branch**: up to resolveAt + outcomeSubmissionGrace only the creator can submit the outcome; after grace - creator or registered market participant. The status changes to AwaitingAppeals, outcomeSubmittedAt and the new appealEpoch are fixed.
- **AwaitingAppeals Window**: Appeals are only allowed within the limits of appealPeriod; the first appeal records targetOutcome, subsequent ones in the same epoch must be with the same target. When the stake threshold is reached, escalation is enabled (escalated=true) and an arbitration case is created.
- **Auto-resolve/overdue-branch**: if in Standard the outcome is not submitted before resolveAt + outcomeSubmissionGrace, anyone can start sendToResolve (or a branch via markFinished) by posting a bond; the market transitions to AwaitingAppeals and linked to the arbitration case.
- **Arbitration branch in Standard**: after the case is finalized, resolveByArbitration is called. If the consensus is below arbitrationFinalConsensusBps, the market remains in AwaitingAppeals with readyToFinalize=true and increased appeal parameters; otherwise, the final outcome is recorded and the market is transferred to Finished.
- **Finished**: markFinished or the arbitration finalize path executes resolveFromRegistry/cancelFromRegistry and then finalizeFees. After this, the market remains in Finished as the terminal status of the register, and economic post-actions go through redeem/claim/retry.

Life cycle of an arbitration case:

- **CaseCreated**: the case is created in ModerationRegistry as CaseKind.Arbitration with createdAt, roundSeconds, committeeSize, maxWeightBps, seed/roster-snapshot and the finalized=false flag.
- **Round scheduling (rotating committee)**: The current round is calculated deterministically from (block.timestamp - createdAt) / roundSeconds. Committee composition and weights are derived from the roster/seed for each round; activity-penalties are applied for missing previous rounds.
- **Voting**: The arbitrator votes via voteOutcome (outcome index including 0 = cancel), the record is stored in voteRecords/arbOutcomeSelection. Repeated voting with the same address on the case is prohibited.

- **Case finalization trigger**: when `round.votes == committeeSize`, `ModerationRegistry` finalizes the case: recalculates the outcome-weight, selects the winning outcome (in case of a tie, tie-break by seed), sets `finalized=true` and `resolvedCommittee`.
- **Market callback**: After finalization, the registry calls `Markets.resolveByArbitration(..., consensusBps)`. Then either the market will end immediately (sufficient consensus), or remain in `AwaitingAppeals` for the next round of appeals/re-arbitration.
- **Payout/slashing stage**: rewards and creator-share are distributed according to on-chain rules (`onArbitrationFinalized`, `distributeCreatorShare`), and for committee members `rewardSlashing/penalizeSlashing` are applied depending on whether their choice matches the final outcome.
- **Important modeling note**: current contracts do not have separate enum states `Evidence` or `Enforced`; their role is played by combinations of case fields (`finalized`, round/vote data, resolved committee) and subsequent callback/claim operations.

8.2.11. Data flows

Data flows:

- **Market creation (create path)**: User Tx `addMarket(...)` → `Markets` (router) → `MarketsLifecycle` (time/outcome validation, transfer `marketFee` to `ModerationRegistry`, write to `MarketsRegistry` with status `Approved`, AMM deployment, case linking) → events `MarketAdded/StatusChanged/MarketAMMDeployed/MarketCaseLinked` → Indexer → Frontend.
- **Trade path**: User Tx `buyDelta/sellDelta` → clone `MarketAMM` (delegatecall in `Core/Fee/Resolution` modules) → status/time/slippage/limits checks → update `q[]`, stake accounting and participant registration in `MarketAMMRegistry` → recording fees in `MarketAMMFees` + recording fee-savings via `FeeDiscountRegistry`/`ClueAccounts` → events `Bought/Sold/FeeRecorded` → Indexer → Frontend.
- **Outcomes, appeals, arbitration (resolution path)**: `submitOutcome` (or overdue path `sendToResolve`) → `MarketsAppeals` (epoch, deposits, threshold) → during escalation `ModerationRegistry`/`Arbitration` (case + committee votes) → callback `resolveByArbitration` → `markFinished` → via the AMM module `resolveFromRegistry/cancelFromRegistry` and `finalizeFees(...)` are executed.
- **Post-finalization and claims (settlement path)**: traders call `redeem` via the AMM module (resolved/cancelled), referrals are claimed through `claimReferralReward`, and unsuccessful transfers are not lost - they remain as pending payout balances and are retried through `retry*` functions (`retryDaoRewardsPayout`, `retryCreatorPayout`, `retryModeratorsPayout`, `retryArbitrationPayout`, `retryBuybackPayout`).
- **DAO rewards lane**: DAO fee share from `MarketAMMFees` → `DAORewardsDistributor.forwardReward(...)` → `DAORegistry.distributeDaoRewards(...)` → pending-reward balances of DAO participants → individual claim entries through the DAO registry.
- **Discounts (discount lane)**: changing stake in `FeeDiscountPool` → `FeeDiscountRegistry.notifyStakeChanged` (reranking top/reserve and slots) → updates seat/index records in `ClueAccounts` → reading `currentDiscountBps(account)` on the AMM on each trade.
- **Governance/upgrade lane**: User Tx `createProposal → vote → finalize` to `DAORegistry` (with grace + executionWindow). The on-chain action is executed upon finalization (without a separate mandatory timelock contract). `set*` calls to `ClueManager`/managers can be made directly from the DAO registry or through the optional `DAOExecutor` (if it is selected as the target executor).

8.2.12. Deploy / Upgrade / Ops

- **Deployment (procedure)**
  - `ClueERC20` with fixed supply is deployed first **500 000 000 CLUE**, which are sent to the address `Owner`.
  - Next, the protocol contracts are deployed in stages: managers, registries, pools, market/lifecycle/appeals, AMM templates and modules. After each stage, connections are configured through `ClueManager` and profile managers (`set*`-parameters/addresses).
  - At the initial distribution stage, the deployer/owner distributes tokens to pool and treasury addresses in accordance with the approved tokenomics (DAO Treasury, Liquidity/MM, Private/Public/Team, etc.).
  - After contract wiring, post-deployment transfer of ownership is performed: contracts with ownership are transferred from deployer to **DAO-governed control** (via `DAOExecutor`), which performs on-chain actions according to the accepted proposal. In other words, the deployer ceases to be the controlling party, and control passes to the DAO.
  - The result of the deployment is the publication of a register of addresses and contract versions for the target mainnet and a public verification of the connectivity of key modules.

- **Upgrade (procedure)**
  - After post-deploy handover, contracts no longer belong to the deployer address; changes are made only through the DAO proposal and voting.
  - When upgrading, a new version of the module/contract is deployed, then the DAO accepts a proposal to update the address in the manager (via `set*` calls).
  - The execution of the approved action occurs in the governance loop (via `DAOExecutor`), so the upgrade has a verifiable on-chain history and does not require manual owner intervention.
  - New markets use updated templates and parameters; existing clones continue to work in their current version, and their migration is formalized by a separate DAO resolution with a separate transition procedure.
- **Operations (Ops)**
  - Continuous monitoring of on-chain market events, AMM, appeals/arbitration, governance and ownership changes; mandatory alerts for `Paused/Unpaused`, changes in critical addresses and abnormal spikes in transaction errors.
  - Operational control of infrastructure: fault-tolerant RPC, health-check of public services, monitoring of latency/indexing lag, runbook for network degradation and fallback data access plan.
  - Indexer and data: block checkpoints, regular backups, the ability to fully replay chain events, periodic reconciliation of the off-chain index with the on-chain source of truth.
  - Release discipline: first testnet/staging, then mainnet. A changelog of addresses and parameters is published along with tx-hash, and for critical changes a pre-announced update window and post-release checks are used.
  - Security of operations: separation of roles, limited access to keys, log of administrative actions, incident response plan (including emergency proposal and communication with the community).

8.2.13. Layered Model and Principles

Below is a summary of the key contracts that together form the core of the protocol: from parameter management and market lifecycle to AMM trading, moderation/arbitration, DAO governance and fee distribution. This is not just a list of addressable components, but a map of their interconnected roles in a single on-chain system.

`Arbitration`
Arbitration module on top of the dispute registry: accepts escalated cases and voluntary market cases, stores committee and weights, records the outcome, distributes rewards to arbitrators, can redirect the creator's share to arbitration upon an upheld appeal, and syncs with the AMM/market registry.

`AutoBurnTreasury`
Burn treasury: accepts CLUE transfers from fee modules and immediately burns the entire balance. The owner can recover any non-CLUE/accidentally sent tokens.

`ClueAccounts`
Central user ledger: unique username for CLUE (DAO reward accruals), profile (CID), accounting of rewards by role (DAO/moderation/referrals/discounts/markets), action counters, status of discount slots and indexes; writes are only allowed to trusted modules.

`ClueERC20`
Base token CLUE (ERC-20 (18 decimals)) with permit, burn and pause; fixed maximum issued to the deployer; metadata via IPFS-URI, pause blocks all transfers.

`ClueManager`
A unified registry of addresses and global parameters: stores links to markets, moderation, arbitration, DAO, stake pools, AMM templates, discount registers, treasuries; checks whether modules belong to the manager and notifies about changes.

`DAOManager`
DAO config: voting boundaries and defaults, quorum, execution window, active participant limit, activity steps, unstake and withdrawal delays; used by the DAO registry and pool.

`DAOPool`
DAO stake pool: staking, locking and unstaking CLUE, accounting for active weight and efficiency by activity, selection of top participants, history of stakes and accruals; business logic is delegated to services.

`DAORegistry`
Registry of governance proposals: stores metadata and target action, voting/execution deadlines, snapshot of staker weight, totals and vote log, quorum and outcome, submission mode; applies penalties for non-participation and triggers the distribution of DAO rewards.

`DAOExecutor`
Executor of governance actions: accepts calls from the DAO and executes targeted `call` on protocol contracts; used as an executor of the approved proposal and post-deploy ownership handover.

`DAORewardsDistributor`
Proxy distributor of DAO rewards: accepts the transfer of tokens and immediately forwards them to the DAO registry for distribution, protected from zero amounts and lack of a registry.

`DAOTreasury`
DAO treasury with geometric decay unlock: calculates the available amount as a share of the remaining budget at a fixed interval, takes into account what has already been withdrawn and protects against over-withdrawal; the owner can preview availability and check status.

`FeeDiscountManager`
Discount system config: 10 slots with bps discount and capacity, withdrawal delay and unstake period for the pool; source of truth for the discount registry and pool.

`FeeDiscountPool`
Trader staking pool for discounts: stores the active stake, locks and unstaking queues, stake history; delegates business logic to the service and requires a configured register of discounts.

`FeeDiscountRegistry`
It ranks stakers and automatically assigns them discount slots: maintains top/reserve tiers, rebuilds ranks when the stake changes, stores snapshots and positions, writes fee savings in ClueAccounts, gives current discounts and ranks.

`MarketAMM`
AMM clone of a specific market: delegates initialization, trading, resolution and finalization of fees to services/modules, stores links to the registry, trading token (`manager.ammTradingToken()`, in the current deployment/configuration (CLUE)), b/fee parameters and internal outcome balances of users, provides checks of rights and market↔AMM connections.

`MarketAMMFXTreasury`
FX treasury for AMM hedging: keeps track of the stablecoin balance for AMM, performs hedge/unhedge through an external pool manager and supports the migration of the stable balance when replacing the AMM clone of the market.

`MarketAMMNoAggregator`
Synthetic NO position aggregator: buys the YES set for all outcomes except the selected one, maintains positions and exposure, checks completeness of coverage and slippage, processes redemptions/cancellations and payouts.

`MarketAMMQuoteHelper`
View helper for complex quotes: selects delta for budget or target payout, takes into account fees and current q-vectors, returns cost/rebate/fee and final pull/payout.

`MarketAMMQuoter`
Static quoter: at AMM, it calculates the buy/sell price with fee, the probabilities of outcomes, can operate based on budget or target payout, uses math/quote helper, does not change the state.

`MarketAMMRegistry`
Market → AMM and Participants Registry: assigns/updates AMM, stores a unique trader count and registration records, allows AMM to register participants and is callable only by the markets service.

`Markets`
Market services router: delegates the life cycle and appeals to modules, stores the state of appeals (deposits, escalations, bonds, escrowed rewards), registers the fee module, writes rewards and referrals in ClueAccounts, relies on registries and managers.

`MarketsAppeals`
Appeals module: accepts CLUE deposits, records the desired outcome, calculates the appeal fee, monitors the escalation threshold for the AMM stake, transfers the dispute to arbitration, supports auto-resolve with the bond and deposit returns.

`MarketsLifecycle`
Life cycle module: checks market parameters, charges marketFee, creates a registry entry and a moderation case, works in the current status model `Approved/AwaitingAppeals/Finished`, deploys AMM clones and registers them.

`MarketsManager`
Configuration of markets and AMM: cost of creation, time limits, maximum outcomes, base trading fee and its split, default AMM parameters (b, win payout, overround, virtual shares), addresses of templates, quoters and factories.

`MarketsRegistry`
Market registry: stores CID, creator, times, status, outcome and outcome CID, associated moderation case; provides pages/ranges, updated only by the markets service.

`Moderation`
Moderation module: creates cases with rewards, accepts committee votes, records status and outcome, distributes rewards to moderators according to weights, writes activity in ClueAccounts; operates via the dispute registry and markets service.

`ModerationRegistry`
Register of disputes and committees: creates moderation and arbitration cases, stores compositions and weights, voting rounds, outcomes, finalization and payout/refund records; assigns deterministic seed committees and checks module rights.

`ModeratorsManager`
Moderation/arbitration config: committee sizes and weights, appeal and escalation parameters, submit/appeal/auto-resolve windows, active moderator limits, activity/slashing, unlock and unstake; source of settings for the registry and moderator pool.

`ModeratorsPool`
Stake pool of moderators/arbitrators: accounting for active stake, locking and unstake, calculation of effective weight taking into account slashing/activity, selection of seed committees, storage of history and CLUE rewards; uses stake/score services.

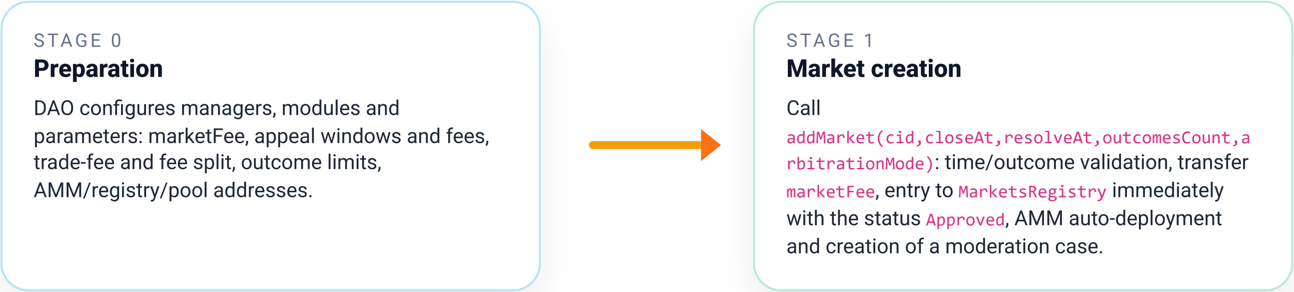
- **No single point of failure:** key operations are performed on-chain, and execution rules are set by smart contracts and DAO governance procedures.
- **Light indexers:** open-source services to speed up data reads/querying (event aggregation, UI cache), without privileges. Any participant can deploy their own query infrastructure or fork the code.
- **Open-source frontend:** static SPA; the code is open, can be deployed locally or on any hosting; does not store private keys.
- **Public RPCs:** use public RPCs of the selected EVM-compatible network; if necessary, the user can specify their own RPC endpoint or local node.
- **Data storage:** market metadata - in IPFS; trades, stakes, fee events, resolution and appeals are strictly on-chain.

8.2.14. Market life cycle (end-to-end)

End-to-end scenario from creation to final settlement. All steps are based on the contracts `Markets.sol`, `MarketsLifecycle.sol`, `MarketsAppeals.sol`, `MarketAMM*.sol`, `MarketAMMFees.sol`, `MarketsRegistry.sol`. Below is a version consistent with the current enum/functions and statuses `Approved → AwaitingAppeals → Finished`.

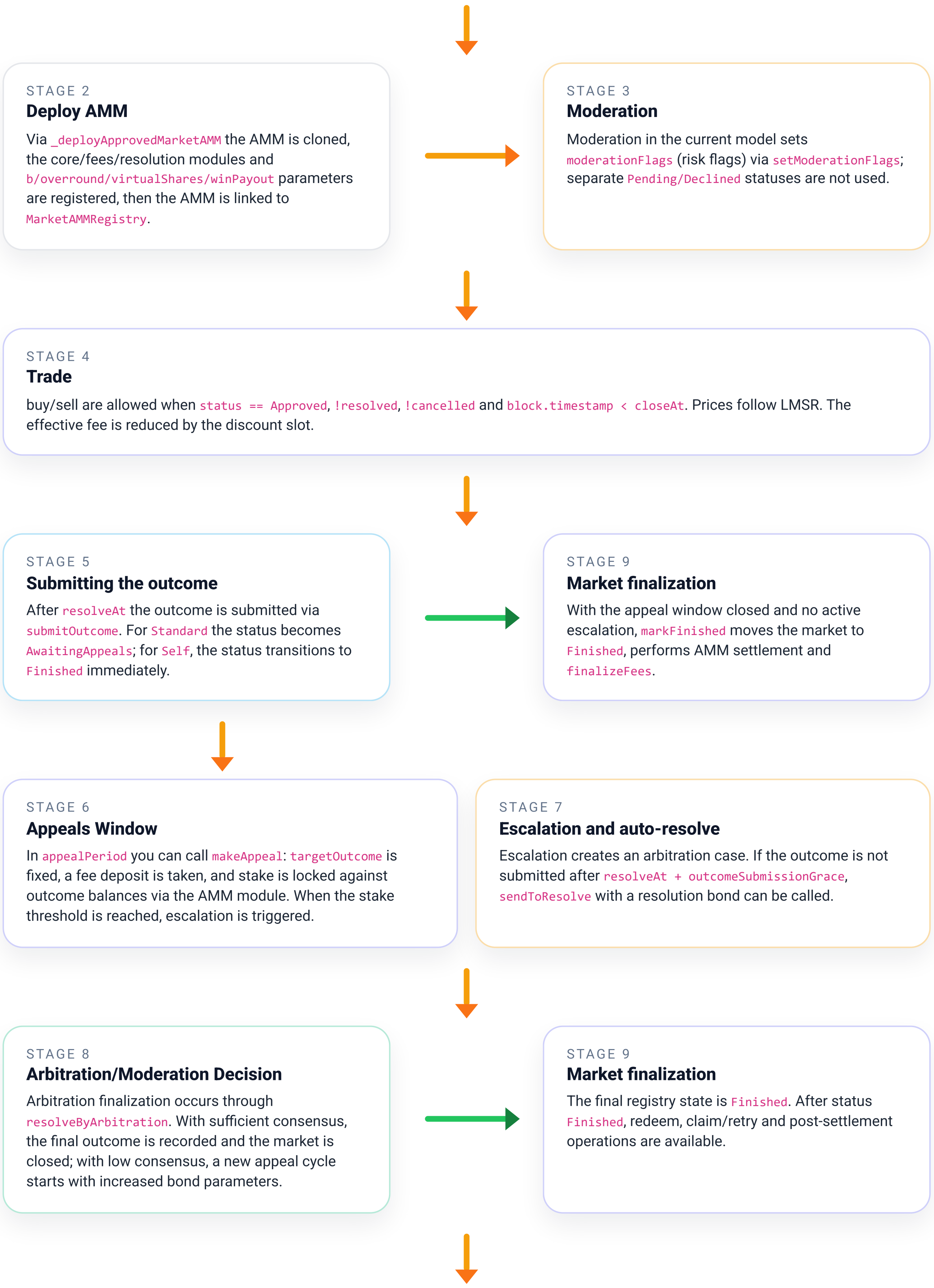
Market Life Cycle Diagram

Full flow:
parameters/roles → `addMarket(Approved)` → AMM → Trade (up to `closeAt`) → submitOutcome → AwaitingAppeals → (appeals/arbitration path if necessary) → Finished → redeem/finalizeFees.



STAGE 0
Preparation
DAO configures managers, modules and parameters: marketFee, appeal windows and fees, trade-fee and fee split, outcome limits, AMM/registry/pool addresses.

STAGE 1
Market creation
Call `addMarket(cid,closeAt,resolveAt,outcomesCount,arbitrationMode)`: time/outcome validation, transfer `marketFee`, entry to `MarketsRegistry` immediately with the status `Approved`, AMM auto-deployment and creation of a moderation case.



STAGE 2

Deploy AMM

Via `_deployApprovedMarketAMM` the AMM is cloned, the core/fees/resolution modules and `b/overround/virtualShares/winPayout` parameters are registered, then the AMM is linked to `MarketAMMRegistry`.

STAGE 3

Moderation

Moderation in the current model sets `moderationFlags` (risk flags) via `setModerationFlags`; separate `Pending/Declined` statuses are not used.

STAGE 4

Trade

buy/sell are allowed when `status == Approved`, `!resolved`, `!cancelled` and `block.timestamp < closeAt`. Prices follow LMSR. The effective fee is reduced by the discount slot.

STAGE 5

Submitting the outcome

After `resolveAt` the outcome is submitted via `submitOutcome`. For `Standard` the status becomes `AwaitingAppeals`; for `Self`, the status transitions to `Finished` immediately.

STAGE 9

Market finalization

With the appeal window closed and no active escalation, `markFinished` moves the market to `Finished`, performs AMM settlement and `finalizeFees`.

STAGE 6

Appeals Window

In `appealPeriod` you can call `makeAppeal`: `targetOutcome` is fixed, a fee deposit is taken, and stake is locked against outcome balances via the AMM module. When the stake threshold is reached, escalation is triggered.

STAGE 7

Escalation and auto-resolve

Escalation creates an arbitration case. If the outcome is not submitted after `resolveAt + outcomeSubmissionGrace`, `sendToResolve` with a resolution bond can be called.

STAGE 8

Arbitration/Moderation Decision

Arbitration finalization occurs through `resolveByArbitration`. With sufficient consensus, the final outcome is recorded and the market is closed; with low consensus, a new appeal cycle starts with increased bond parameters.

STAGE 9

Market finalization

The final registry state is `Finished`. After status `Finished`, redeem, claim/retry and post-settlement operations are available.

STAGE 10

Redemptions via the AMM module

In the resolved branch, redeem pays according to the winning outcome; in the canceled branch, the refund is proportional to the user's share. Operations burn internal AMM outcome positions.

STAGE 11

Distribution of trading fees

After settlement, `finalizeFees` is called: split by `ammFeeDistributionBps` (buyback/creator/moderation/referral-or-DAO), plus retry mechanics for unsuccessful transfers.

Volume → fees → burn/treasury & rewards → stake roles → reliable moderation/arbitration → more markets → new volume

Roles and key parameters

- `ClueManager` **(wiring)**: stores the addresses of the markets router, lifecycle/appeals modules, registries, moderation/arbitration, AMM modules, treasuries and DAO governance. In the current governance model, ownership is held by the DAO (usually via `DAORegistry`/`DAOExecutor`). Unbound or unconfigured modules give `ModuleNotConfigured/ModuleManagerMismatch`.
- `MarketsManager` **(market parameters)**: sets `marketFee`, `minExpiryDuration`, `minExpiryToEventGap`, `maxOutcomesCount`, `ammTradeFeeBps`, `ammFeeDistributionBps`, `outcomeSubmissionBondBps`, as well as AMM defaults (`b/overround/virtualShares/winPayout`).
- `ModeratorsManager` **(disputes/arbitration)**: sets `appealPeriod`, `outcomeSubmissionGrace`, `autoResolveGrace`, `appealFeeParams`, `minAppealBond`, `appealEscalationThresholdBps`, `arbitrationFinalConsensusBps`, `appealBondEscalationStepBps`.
- `MarketsRegistry` **(source of truth)**: in the current core flow, only the `Approved`, `AwaitingAppeals`, `Finished` statuses are used (without `Pending/Declined`).
- **AMM state**: outcome positions are maintained in the internal AMM state (`outcomeBalances`, `q[]`, treasury), and the trading token is taken from `manager.ammTradingToken()` (in the current protocol profile - `CLUE` as trading+utility token).
- **Global denominator**: `bpsDenom = 100 000` for fee, split and escalation thresholds.

Stage 0 - preparation

- The DAO owner configures manager wiring and manager parameters before starting trading.
- Moderators/arbitrators accumulate stake weight in the pool; penalties in the core flow are score/slashing modifiers, not automatic on-chain stake burning.
- Traders can claim fee-discount slots via `FeeDiscountPool/Registry`. The discount reduces the effective fee, but does not change the proportion of the fee split.

Stage 1 - creating a market

1. Call `Markets.addMarket(cid, closeAt, resolveAt, outcomesCount, arbitrationMode)`.
2. Checks: `closeAt > now`; `closeAt ≥ now + minExpiryDuration`; `resolveAt > closeAt`; `resolveAt ≥ closeAt + minExpiryToEventGap`; limiting market duration; `2 ≤ outcomesCount ≤ maxOutcomesCount`; moderation modules and registries are configured.
3. **marketFee** (CLUE) is transferred to `moderationRegistry`.
4. Creating an entry in `MarketsRegistry` immediately with status **Approved**; the `MarketCreated` counter is incremented.
5. Auto-deployment of the AMM clone and its linking to `IMarketAMMRegistry`.
6. Auto-creation of a moderation case via `IModerationCaseFactory`.
7. Events: `MarketAdded` (records parameters), `MarketAMMDeployed`, `MarketCaseLinked` (links the market and moderation case). `MarketsRegistry` stores: `cid`, `creator`, timestamps, `outcomesCount`, `arbitrationMode`, `moderationCaseId`.

Stage 2 - moderation (flags model)

- In the current implementation, moderation does not switch market status between Pending/Declined.
- Through `setModerationFlags` only the risk bitmask `moderationFlags` is updated in the registry.
- The creator share is redirected to arbitration when the target outcome is confirmed.

Stage 3 - AMM deployment (for Approved status)

- `_deployApprovedMarketAMM` clones the AMM template and links it to `IMarketAMMRegistry`.
- Parameters from `marketsManager.ammDefaults()`: `bWad`, `winPayout`, `overround`, `virtualShares`, `coreModule`.
- `feeWad = currentTradeFeeWad()`; for an address without a discount it is `ammTradeFeeBps / bpsDenom * 1e18`. In AMM, core/fees/resolution modules and burn/fee fallback treasury are configured.
- Checks: manager/registry compliance, market↔AMM link, valid modules/token/parameters.

Stage 4 - trading (up to closeAt)

Allowed if: status **Approved**, AMM bound, `!resolved`, `!cancelled`, current time is less than `closeAt`.

- **Purchase** `buyDelta(outcome, delta, maxCost, referral, deadline)`: updates `q[]`, treasury and internal balance of the outcome position.
- **Sale** `sellDelta(outcome, delta, minOut, referral, deadline)`: reduces the position and pays a rebate minus fee.
- **Boundary exponent**: checking `q[outcome]+delta` vs `EXP_INPUT_LIMIT_WAD` relative to `b`.
- Both operations update `q[]`, `totalRealShares`, `aggregateStakeWad`, register a participant via `ammRegistry.registerParticipant` and log events `Bought` / `Sold` with a full liquidity vector and `b` value before/after the transaction.
- Additional checks: deadline/slippage, same-block execution guard (exception - aggregator address `ammNoAggregator`).

Stage 5 - submission of the outcome

- `submitOutcome` is available after `resolveAt`, with status **Approved** and if the outcome has not yet been submitted.
- OutcomeIndex: `0` - cancel; otherwise `1..outcomesCount`.
- **Self mode**: creator only; when submitting, the status is immediately `Finished` and settlement and fee finalization are executed.
- **Standard mode**: in a grace window, only the creator can submit; after grace - creator or registered market participant. Status → **AwaitingAppeals**, `outcomeSubmittedAt` is recorded, and the appeal epoch starts.
- Before updating, the contract checks that `arbitrationContract` is set to ensure appeals are possible. All status/outcome changes are accompanied by `OutcomeSubmitted` and `StatusChanged` events.

Stage 6 - Appeals Window

- Duration: `appealPeriod` seconds from `outcomeSubmittedAt`.
- **makeAppeal**: Anyone can appeal with `desiredOutcome` selected (≠ outcome). Stake is locked from AMM outcome balances (`outcomeBalanceOf`): for cancel appeals, stake is aggregated across all outcomes, otherwise for the selected outcome.
- The appeal fee is calculated as `max(stakeWad × feeBps / bpsDenom, stakeAgainst × floorBps / bpsDenom)`, transferred to `rewardPool`, the deposit is credited to the participant.
- `targetOutcome` is fixed by the first appeal; subsequent appeals must match.
- **Escalation threshold**: compares `appealStakeTotal` against `stakeAgainstWad` (using the baseline snapshot, or if unavailable, by the current `q[]`): $appealStakeTotal \times bpsDenom \geq stakeAgainstWad \times thresholdBps$. If the threshold is met, the dispute escalates to arbitration.
- Note: if `stakeBaselineWad` is not yet committed, it is snapshotted from `q[]` when the appeal branch is opened and used for threshold calculations.
- Each appellant is stored in `AppealState.participants[address]` with fields `lastAppealEpoch`, `depositEpoch`, `depositAmount`. The deposit is returned only if `refundMode` is active and the epoch is valid.

Stage 7 - escalation and auto-resolve

- Appeal escalation creates an arbitration case, the market is linked to `arbitrationCaseId`, the `escalated/wasArbitrated` flags are set to true.
- Auto-resolve: if the outcome is not submitted and `resolveAt + outcomeSubmissionGrace` has passed, anyone can call `sendToResolve`; the bond size is calculated based on the appeal parameters with the lower limit `minAppealBond` (CLUE), an arbitration case is created, the status transitions to **AwaitingAppeals** if necessary.

- Bond refund: for resolution bond, refund-path is used with clearAppeals and in arbitration finalization (resolveByArbitration/_distributeAppealBond). If there is insufficient balance, a partial refund is possible with the balance recorded in pending and a separate claim entry via claimPendingAutoResolveBond.
- When escalating, AppealEscalated is emitted, and when auto-resolve is called AutoResolveTriggered + AutoResolveBondPosted. Further outcome finalization is executed via resolveByArbitration.

Stage 8 - Arbitration/Moderation Decision

- The arbitration process is finalized by calling resolveByArbitration(id,outcomeIndex,caseId,consensusBps) (call from moderationRegistry).
- If the average consensus is below arbitrationFinalConsensusBps, the market remains in AwaitingAppeals, readyToFinalize is set to true, and the appeal fee/bond parameters are increased by appealBondEscalationStepBps.
- If the consensus is sufficient, the final outcome is recorded, if necessary, redirectCreatorShare is enabled, then settlement and fee finalization are executed.
- redirectCreatorShare is activated only if targetOutcome matches the final outcome of the arbitration.

Stage 9 - market finalization

- markFinished is available when status is **AwaitingAppeals** (window closed or appeals cleared) or **Approved**; active escalation prevents finalization.
- Status → **Finished**; appeal metrics are reset.
- AMM: if there is an outcome and the AMM is not closed - cancelFromRegistry (if outcomeIndex=0) or resolveFromRegistry (winningOutcome = outcomeIndex-1), then finalizeFees(redirectCreatorShare, arbitrationCaseId).
- The MarketResolved counter is incremented for the creator.
- If the market was in **AwaitingAppeals** and has accumulated rewardPool, upon completion without active escalation, refundMode is enabled, allowing appellants to claim refunds; after the pool is empty, refundMode is disabled.
- After AMM settlement, the finalizeFees call is mandatory: without it, the fee will remain in the module and will not be distributed. If redirectCreatorShare is active, the absence of arbitrationCaseId reverts the transaction at the lifecycle/fees layer (MissingArbitrationCase/InvalidArbitrationCase).

Stage 10 – payouts via the AMM module

- **Resolved**: holders of winning AMM outcome positions redeem and receive a payout: $payout = amountWinTokens \times winPayoutPerToken$, where $winPayoutPerToken = treasury/q_{winningOutcome}$.
- **Cancelled**: users burn all their tokens, receive a treasury share in proportion to $userShares/totalRealShares$ (if they hold all shares, they can receive the entire balance).
- All payouts go through MarketAMMResolutionModule: upon cancellation, it is checked that the user has an internal outcome balance; upon resolution, only winning positions are burned. Any invalid attempts (without balance or with zero treasury) revert.
- redeem() additionally tries to claim arbitration bond and appeal refund (if available).

Stage 11 - Distribution of trading fees (after finalizeFees)

- Shares from ammFeeDistributionBps: **burn/fee-treasury**, **creator**, **moderation**, **referral/DAO**. Unallocated balances go to burn allocation.
- **Referral**: if referral != 0, their share accumulates in referralRewards[referral]; otherwise their share is added to feeDaoRewardsTokens.
- **RedirectCreatorShare**: if the appeal is successful, the creator's share goes to the arbitration case via Arbitration.distributeCreatorShare (call via the IArbitrationRewards interface).
- **DAO Rewards**: payment via daoRewardsDistributor.forwardReward; if unsuccessful, retrying via retryDaoRewardsPayout is available.
- All payments are made from the fee module through secure transfer/approve paths (including forward to DAO distributor). If the receiving module does not accept the transfer, the funds remain in feeDaoRewardsTokens until the call is successful.
- For unsuccessful transfers, retry functions are available: retryDaoRewardsPayout, retryArbitrationPayout, retryCreatorPayout, retryModeratorsPayout, retryBuybackPayout.

LMSR formulas (price calculated by `MarketAMMMath`)

- Cost function: $C(\mathbf{q}, b) = b \cdot \ln\left(\sum_i e^{q_i/b}\right)$.
- Buy price for outcome delta k: $\text{cost} = C(q + \Delta_k, b) - C(q, b)$.
- Sell rebate for delta k: $\text{rebate} = C(q, b) - C(q - \Delta_k, b)$.
- Marginal price of outcome k: $p_k = \frac{e^{q_k/b}}{\sum_i e^{q_i/b}}$.
- Dynamic `b`: $b = \max\left(baseB,\ \alpha \cdot liquidityProxy\right)$, where $\alpha = \frac{1+overround}{n \cdot \ln n}$.
- The `quoteBuy`/`quoteSell` functions only work with 18-decimal precision (wad). In case of insufficient liquidity (`q[outcome] < delta`) or zero `b`, operations revert with `InvalidB`/`NegativeLiquidity` from the library.

Fees and discounts

- Base rate: `baseFeeBps = ammTradeFeeBps`.
- Discount: `discountBps = feeDiscountRegistry.currentDiscountBps(user)` (in the code it is capped by `bpsDenom`).
- Effective rate: $feeRate_{eff} = feeRate_{base} \times (bpsDenom - discountBps)/bpsDenom$.
- Transaction fee: `buy: fee = cost × rate / (1 - rate)`, `sell: fee = rebate × rate`.
- Savings from a discount: $savedWad = max(0, baseFeeWad - feeWad)$, where for buy `baseFeeWad` is calculated using the same gross fee formula based on `cost` and the base rate, and for sell - as `rebate × baseRateWad`.
- The fee is transferred to `MarketAMMFees.recordFee`, where it is converted to token amounts, taking into account the decimal places of the trading token (in the current deployment/configuration (CLUE)). If `feesModule` is not configured, buy/sell operations revert (`ModuleNotConfigured`).

Appeal refunds

- If `refundMode=true`, a member with an active epoch deposit can call `claimAppealRefund`: receives their `depositAmount` (CLUE), `rewardPool` is decreased.
- When `rewardPool` is depleted, `refundMode` is turned off and `targetOutcome` is reset.
- Upon successful refund, `AppealRefundClaimed` is emitted, and the participant's deposit is reset to zero. If there are not enough funds or the epoch does not match, the operation is aborted (revert `RefundUnavailable`).

Transitioning back to **Approved** from **AwaitingAppeals** does not occur in a normal flow: finalization leads to **Finished**. After status **Finished** the market can no longer be traded or appealed.

Brief overview of CLUE flows

- **CLUE as utility token**: market creation fee, appeal fee and auto-resolve bond, role staking flows (DAO/moderation/discount) and reward accounting in `ClueAccounts`.
- **CLUE as trading token (current profile)**: in the current configuration `manager.ammTradingToken() = CLUE`, so trading fees go to CLUE and are split into burn/treasury, creator / arbitration, moderation, referral/DAO. Changing the trading token is possible only through the owner/governance path.
- **CLUE as a settlement token**: trading fees → split (burn/treasury, creator / arbitration, moderation, referral/DAO); payments to winning-position holders/refunds upon cancel; push/pull via AMM.
- CLUE reward flows are captured in events and are accessible via `pendingReward`/`consumeReferralReward` in `ClueAccounts`. Protocol trading token flows (in the current profile - `CLUE`) pass either through the AMM itself (buy/sell/redeem) or through the `MarketAMMFees` module.

8.3. Flows of Fees, Roles, and Funds

- **Protocol revenue sources:** AMM trading fees in `manager.ammTradingToken()` (in the current protocol profile this is `CLUE`; changing the address is only possible with an owner/governance call), market creation fee (`marketFee`, taken in CLUE), as well as appeal bonds and resolution bond (also in CLUE).
- **Trading fee distribution:** `MarketAMMFees` accumulates fees on fee allocations and when `finalizeFees` sends shares to `buybackTreasury` (fallback: `feeTreasury`), `creator/moderation` and referral/DAO branch. With `redirectCreatorShare` the creator share goes to `Arbitration.distributeCreatorShare`; referral rewards are claimed through `claimReferralReward`, and unsuccessful transfers remain in pending and are closed through `retry*` functions.
- **FX treasury lane in fund flow:** when FX mode is enabled, AMM may route part of buy-side net through `MarketAMMFXTreasury.hedgeToStable` and pull liquidity back via `redeemTradingToken` during sell/payout pressure. This changes the treasury composition (trading-token ↔ stable), while fee allocations and split logic in `MarketAMMFees` remain deterministic.
- **Synthetic NO execution path:** `MarketAMMNoAggregator` composes NO exposure through full YES-leg baskets (`N-1` outcomes), applies aggregate slippage checks, and settles through snapshot-based redemption/cancel paths; fee accrual still follows normal AMM execution routes.
- **Discounts:** stake in `FeeDiscountPool` affects `discountBps` from `FeeDiscountRegistry`, reducing the effective fee rate without changing the bps split; savings are recorded as `RewardKind.FeeDiscount`.
- **Penalties and quality control:** in the core flow, penalties for moderators/arbitrators are changes in activity/slashing coefficients in pools; direct token slashing of moderators is available via `ModeratorsPool.slashStake` and is only called by DAO governance (`daoRegistry`/`daoExecutor`). `ClueAccounts` stores the reward ledger and counters of actions/penalty events, and not the stake itself.

8.4. Governance Controls and Upgrades

- **DAO as owner**: in the target configuration, owner rights are consolidated under DAO governance (usually `DAORegistry` + `DAOExecutor`), so critical `set*`/`upgrade` actions go through governance.
- **Ownership and access**: `Ownable` explicitly applied to `ClueManager` and `ClueERC20`. Some services (for example, `Markets`, `MarketsLifecycle`, `Moderation`) are not `Ownable`, but work through hard role/check modifiers and contract-registry wiring.
- **Governance execution model**: proposal is created/voted in `DAORegistry`, and with `finalize` the target action is executed on-chain via `target.call(actionCalldata)`. Timings are set by parameters `votingDeadline + resolveGracePeriod + executionWindow`. There is no separate timelock contract in the current code.
- **Changing settings**: `set*` in managers (`MarketsManager`, `DAOManager`, `ModeratorsManager`, `FeeDiscountManager`) available are restricted to `onlyManagerOwner` (owner `ClueManager` or `ClueManager` itself), and changes are captured by events like `SettingsChanged`.
- **Module upgrades**: the new version is deployed and the address is updated through the manager-layer/DAO action. New markets use a new configuration, and already created AMM clones are not upgraded automatically - they require manual migration.
- **DAOExecutor**: optional execution adapter; if `daoRegistry` is configured, only the registry can call it, otherwise the fallback caller is the manager owner.
- **Emergency actions**: `pause/unpause` in `ClueERC20` are available to the owner address; reconfiguration of critical addresses (`buyback/fee/registry/module`) occurs through on-chain governance and leaves an event log.

9. Legal Aspects and Compliance

In this section we set a simple and clear framework: **CLUE is a protocol for on-chain markets**, and not a centralized service with an “order book” and a single operator. Trading in CLUE is structured as follows: **AMM-based pool model (LMSR)**: participants interact with the liquidity of the pool according to the rules of the smart contract, **no limit orders** and without manual matching by a centralized operator/service. An open-source reference interface can exist separately from the protocol and be hosted on an independent domain.

Below is how this is interpreted in compliance and the distribution of responsibilities between the protocol, interfaces and operators.

9.1. Regulatory principles

CLUE follows a **protocol-first** model: there is a basic protocol layer (contracts, rules, DAO), and there is a separate interface layer (web interfaces, APIs, local applications). This helps to correctly describe the product in different jurisdictions.

- **Protocol as infrastructure:** CLUE sets the rules for the execution of transactions and settlements by smart-contract code, and not through manual operator decisions.
- **Pool-based non-custodial trading via AMM:** participants trade with a liquidity pool using a mathematical model, without a limit order book and without a centralized matching engine.
- **Hedging use-case by design:** outcome positions are tradable before resolution, so users can hedge, re-hedge and rebalance risk exposure instead of waiting for a binary end-state payout.
- **Non-custodial access:** the user controls their own wallet and signature of transactions; the protocol does not store users' private keys.
- **Separation of interfaces:** The web/app layer can be launched by different teams and on different domains, including an open-source community frontend.
- **DAO governance:** changes to parameters go through on-chain governance, which increases the predictability and transparency of the rules.
- **Local requirements at the interface/operator level:** If the interface operator is subject to licensing, registration, marketing or access restrictions, it will comply with those requirements in its jurisdiction.
- **Proper token disclosure:** the CLUE token is described as a utility/governance component of the ecosystem; any language about “guaranteed returns” is excluded.

Legal classification boundary: what CLUE is not

For legal qualification, it is equally important to describe not only what the protocol does, but also what it does not do by design. In CLUE, the core mechanics are built around trading transferable outcome tokens and on-chain price discovery, not around a bookmaker model with operator liability.

- **CLUE is not a bookmaker model.** The protocol does not accept wagers into an operator balance sheet and does not act as a counterparty that takes on users’ loss risk. Transactions are executed by participants under AMM rules.
  Consequence: there is no bookmaker/house risk model.
- **CLUE is not a gambling operator paying winnings.** There are no fixed odds, no promised payout table, and no operator-side redistribution from losing positions to winning positions. Economic result is formed through trading position dynamics.
  Consequence: the model is asset-trading based, not betting-based.
- **CLUE is not a custodial financial intermediary.** Users keep wallet control, while contracts execute deterministic rules; neither team nor DAO can arbitrarily move user balances.
  Consequence: the protocol does not perform custodial intermediation.
- **CLUE does not set odds by operator discretion.** Outcome-token prices are formed by market interaction and AMM mathematics, not by a centralized quote desk.
  Consequence: a key hallmark of classic bookmaker activity is absent.
- **CLUE implements trading positions, not wagers.** A user buys an outcome-token with a live market price and may exit prior to resolution.
  Consequence: the economic action is opening/closing a market position.
- **CLUE is not a betting platform.** Market participants can dynamically manage exposure through entry, exit, hedge and re-hedge actions while the market is live.
  Consequence: user behavior corresponds to risk management on tradable assets, not to one-shot wager placement.

- **Core activity happens before event resolution.** Outcome-tokens are tradable pre-resolution, and users can rebalance or close exposure without waiting for the final state.
  Consequence: protocol function is price discovery, not ex post winner-payout distribution.
- **Settlement is an asset-state transition, not a bookmaker payout event.** Resolution updates market state and redemption logic under code-defined rules, without discretionary operator settlement.
  Consequence: there is no centralized bet settlement engine.
- **CLUE is infrastructure, not a centralized market operator.** Smart contracts run autonomously, users create markets, and governance changes pass via DAO procedures.
  Consequence: the system is closer to open financial infrastructure than to a single-operator service model.
- **CLUE does not promise profit and does not frame participation as a game.** There is no guaranteed return; outcomes depend on market behavior and participant decisions.
  Consequence: there is no gambling-style behavioral inducement encoded in protocol rules.

Additional exclusions relevant for legal classification

- **No centralized order book and no operator matching desk.** Execution follows AMM state transitions in smart contracts, rather than discretionary order matching.
  Consequence: execution logic is infrastructural and code-driven.
- **No operator balance-sheet exposure ("house book") to user losses.** The protocol does not maintain a house balance that systematically profits from user losses.
  Consequence: there is no classic “house edge vs players” architecture.
- **No discretionary trade approval by a central admin.** Trades are accepted or rejected by deterministic on-chain checks (status, time windows, limits).
  Consequence: transaction admissibility is rule-based, not manager-based.
- **No off-protocol settlement committee with privileged final say.** Outcome and dispute pathways are formalized on-chain through published contract logic.
  Consequence: result finalization is verifiable and auditable by any observer.
- **No unilateral market shutdown by a web interface operator.** Interface access can change, but market state transitions remain bound to contract permissions.
  Consequence: UI operation and protocol operation are legally and technically separable layers.
- **No unilateral fee rewrite by the team.** Core parameter changes are routed through DAO governance procedures.
  Consequence: fee policy is governance-constrained, not centrally mutable at will.
- **No mandatory dependence on a single frontend or API.** Multiple interfaces can connect to the same protocol state.
  Consequence: access is not equivalent to one operator-controlled interface/service.
- **No protocol-level user account custody model.** Wallet signatures authorize operations; private keys are not deposited into protocol control.
  Consequence: users remain self-custodial principals of transaction intent.
- **No mandatory centralized onboarding at protocol layer.** Compliance onboarding, where required, belongs to the interface/operator responsibility scope.
  Consequence: regulatory obligations are allocated by service role, not conflated with base protocol logic.
- **No guaranteed capital protection or fixed return profile.** Economic outcome is market-dependent and position-dependent.
  Consequence: protocol communication is utility and market-risk based, not promise-based.
- **No operator-authored odds table as a source of payoff rights.** Pricing and implied probabilities emerge from market interaction under AMM mathematics.
  Consequence: payout logic does not rely on bookmaker odds-setting discretion.
- **No jackpot/prize-pool game mechanics managed by a central host.** The protocol focuses on transferable outcome-asset positions and settlement rules.
  Consequence: participation is structured as market exposure management, not game-cycle participation.
- **No rehypothecation of user funds by protocol operator.** User funds are not centrally re-used as operator collateral.
  Consequence: the protocol is not acting as a balance-sheet intermediary.
- **No privileged off-chain rewrite of historical market events.** Trade and settlement history is anchored in chain data and event logs.
  Consequence: the risk of after-the-fact record manipulation is structurally reduced.
- **No dependency on continuous team operation for core execution.** Contracts can continue to process valid calls without day-to-day operator intervention.
  Consequence: the system aligns with open protocol infrastructure rather than managed service operation.

9.2. KYC/AML

CLUE has a multi-level approach to KYC/AML. At the protocol level, there is no centralized onboarding and a "KYC" form. At the interface/operator level, requirements may be mandatory, depending on the law and the business model of a particular operator.

- **Protocol level:** smart contracts execute on-chain rules equally for all transactions and do not maintain a centralized database of KYC profiles.
- **Interface/operator level:** KYC/AML, sanctions screening/control, geo-restrictions and internal monitoring procedures are carried out where required by local law.
- **RBA approach:** For high-risk scenarios, operators can apply enhanced screening (EDD), limits and additional procedures.
- **Travel Rule and reporting:** if the service qualifies as a VASP/equivalent, the responsibilities for data exchange and reporting are performed by that operator/service.
- **Transparent division of roles:** CLUE as a protocol does not replace the compliance obligations of a specific interface/operator. Each operator is responsible for complying with requirements applicable in its jurisdiction and to its users.

9.3. Structure

To avoid confusion, CLUE is described as a multi-layer system:

- **Protocol layer (CLUE):** smart contracts, AMM logic, on-chain settlements, governance and open source code.
- **Interface layer:** web/app frontends, indexers and API adapters that help you interact more conveniently with the protocol.
- **Operator layer:** independent teams that launch interfaces under their own brand and in their own jurisdiction.
- **Governance layer (DAO):** collective change of protocol parameters through formalized procedures and on-chain voting.
- **Documents and policies:** Each interface operator must have its own ToS, Privacy and, if necessary, AML/sanctions procedures with a clear disclosure of the scope of responsibilities.
- **Bottom line:** the protocol remains a neutral technological basis, and operational and regulatory responsibilities are concentrated on the specific interface/operator side.

9.4. Disclaimers

- **About the nature of CLUE:** CLUE is a pool-based AMM trading protocol without limit orders and with pre-resolution position hedging. This is not a single centralized betting service or a betting platform.
- **About interfaces:** access interfaces can be different and are legally distinct from the protocol itself.
- **About the scope of responsibility:** The operator of a particular interface is responsible for the legal, licensing and compliance requirements that apply to it.
- **Informational nature of the materials:** The whitepaper does not constitute investment, legal or tax advice.
- **About the token:** CLUE's utility/governance functions should not be interpreted as an unconditional promise of profit or share in the company.
- **About jurisdictions:** It is important for the user to take into account local restrictions, sanctions requirements, and tax rules.
- **About the risks of use:** software and infrastructure are provided "as is", with technological and market risks characteristic of public blockchain systems.

It is recommended that the final legal formulations be further validated for each target jurisdiction with specialized consultants. The basic position of the section remains unchanged: CLUE is an infrastructure protocol with on-chain rules, and not a centralized betting platform.

10. Risks and Mitigation Measures

10.1. Technical

- **Smart contract vulnerabilities:** at the DEV stage, an internal audit and verification of key components of the protocol were carried out; no critical vulnerabilities were identified. After attracting funding, an independent external audit is planned with the publication of a report, severity analysis and remediation plan.
- **Infrastructure failures:** backup RPC/indexer providers, open-source self-hostable frontend/indexer repositories, and automatic failover to backup infrastructure.
- **MEV/front-running:** recommendations for private RPC endpoints / private transaction relays, protection of sensitive operations, anomaly analysis and alerts.
- **FX execution-path dependency:** with FX enabled, hedge/unhedge depends on pool-manager connectivity, route parameters and available depth; stressed markets can increase revert probability. Mitigation: registry-bound AMM checks, validated pool defaults, and fail-closed slippage bounds (`minOut/maxIn`).
- **Synthetic NO multi-leg complexity:** NO aggregation executes across `N-1` legs and is more sensitive to gas spikes and leg-level liquidity gaps than a single-leg trade. Mitigation: full-coverage, dedupe and equal-leg validation, aggregate slippage checks, and deterministic snapshot settlement in `MarketAMMNoAggregator`.
- **Resolution data-source integrity (without privileged oracle dependency):** multiple data sources, fallback logic, and discrepancy monitoring.
- **Keys and access:** multisig/DAO governance delay controls, role minimization, key-rotation procedures.

10.2. Economic

- **High fee rate for active trading:** basic fee 3% may be perceived as high relative to some DeFi/Prediction alternatives, which creates a risk of volume outflow for high-frequency traders. Mitigation: CLUE uses AMM-based trading through LMSR without limit orders, which provides more flexible entry/exit mechanics and potential profit scenarios compared to a number of CPMM approaches. Additionally, there are staking-based discounts that reduce the effective rate for active participants.
- **FX basis and hedge-cost drift:** when the FX lane is used, differences between trading-token moves and stable-lane execution can reduce net realized edge during high volatility windows. Mitigation: bounded hedge execution (`minOut/maxIn`), configurable FX defaults, and DAO parameter tuning as liquidity regimes change.
- **Synthetic NO cost compounding:** NO exposure in multi-outcome markets is assembled from multiple YES legs, so aggregate slippage/fees can exceed single-leg intuition in thin liquidity conditions. Mitigation: strict user slippage caps, quote-preview before execution, and preference for deeper markets.
- **Dependence on the referral mechanism and creator economy:** If the organic influx of users and market creators is below expectations, the growth flywheel may spin slower than the target rate. Mitigation: simplification of referral onboarding, transparent reward model and systemic support for creators who create high-quality markets and long-term user demand.
- **Uncertainty regarding Public Distribution before TGE:** while the unlock parameters for Public Distribution remain *TBD*, this creates additional uncertainty for price expectations and the assessment of future supply. Mitigation: as transparent and early communication as possible on the unlock scenario, TGE timing and the expected impact of distribution parameters on market dynamics.
- **Token volatility:** deflationary burn, DAO setting of fees/discounts, supply and unlock control.
- **Liquidity:** market making and LP incentive programs, priority liquidity pools on DEXs, staged listings on CEX.
- **Manipulation and collusion of roles:** rotation of moderators/arbitrators, slashing, public decision logs, stake weight and randomized committee selection.
- **Referral/creator abuse:** ban on self-referral, limits, anti-wash trading rules, disabling rewards for repeated violations.
- **Spreads and depth:** liquidity and price alerts, incentive rebalancing.

10.3. Regulatory

- **Regulatory changes across jurisdictions:** modular compliance framework, consultations, adjustments to the availability of functions and interfaces.
- **KYC/AML/sanctions (for interface operators/services):** risk-based approach, checks against sanctions lists, escalation procedures and an auditable decision log.
- **Licensing (if necessary):** assessment of VASP/PSP/EMI requirements, preparation of a package of documents, updating public policies.

10.4. Operational

- **Abuse/farming:** anti-abuse metrics, limits, slashing, cross-account monitoring.
- **Cross-chain/interop risk (if bridge integrations are enabled):** limits and delays on bridges, multisig and DAO parameter control, anomaly monitoring.
- **Third-party/provider failures (RPC, indexing services, hosting):** backup providers, automatic switching, SLO/alerts.
- **Communications and incidents:** incident notification procedures, status page, post-mortem and public incident reports to the DAO/community.

11. Team & Contributors

- **Protocol Architect — Andrei Nistor:**
  Full-Stack Engineer. 15+ years in IT spanning full-stack engineering, application security, and economic modeling, with a focus on secure system architecture and incentive-aligned protocol design.
- **Co-founder — Olga Doruc:**
  Leads cross-border communications and partnership development, facilitating international outreach, strategic discussions, and long-term relationship management with ecosystem participants, partners, and stakeholders.

12. Conclusion

CLUE is a rare case when decentralization looks not like a slogan, but like an operational design: on-chain rules, a high degree of decentralization at the protocol layer, a burn share is applied to AMM trading fees, and stake is built into every role. Here, the value of the token is not promised “at some future date”, it is mechanically created by trading volume: the more trading volume, the less supply, and this cannot be reversed by team decision. This combination of code, incentives, and transparent governance through the DAO is a level of integrity that prediction markets have long lacked.

CLUE is what Web3 protocols so often lack: mature tokenomics with a good balance, treasury discipline through governance execution delays and phased unlock schedules, and most importantly, a flywheel that is powered by user actions, not subsidies. If these principles continue into production, CLUE has the potential to become the default infrastructure for prediction markets: fair, censorship-resistant, and economically self-sustaining. This is exactly the kind of ending you would expect from a system that relies on verifiability rather than promises.