

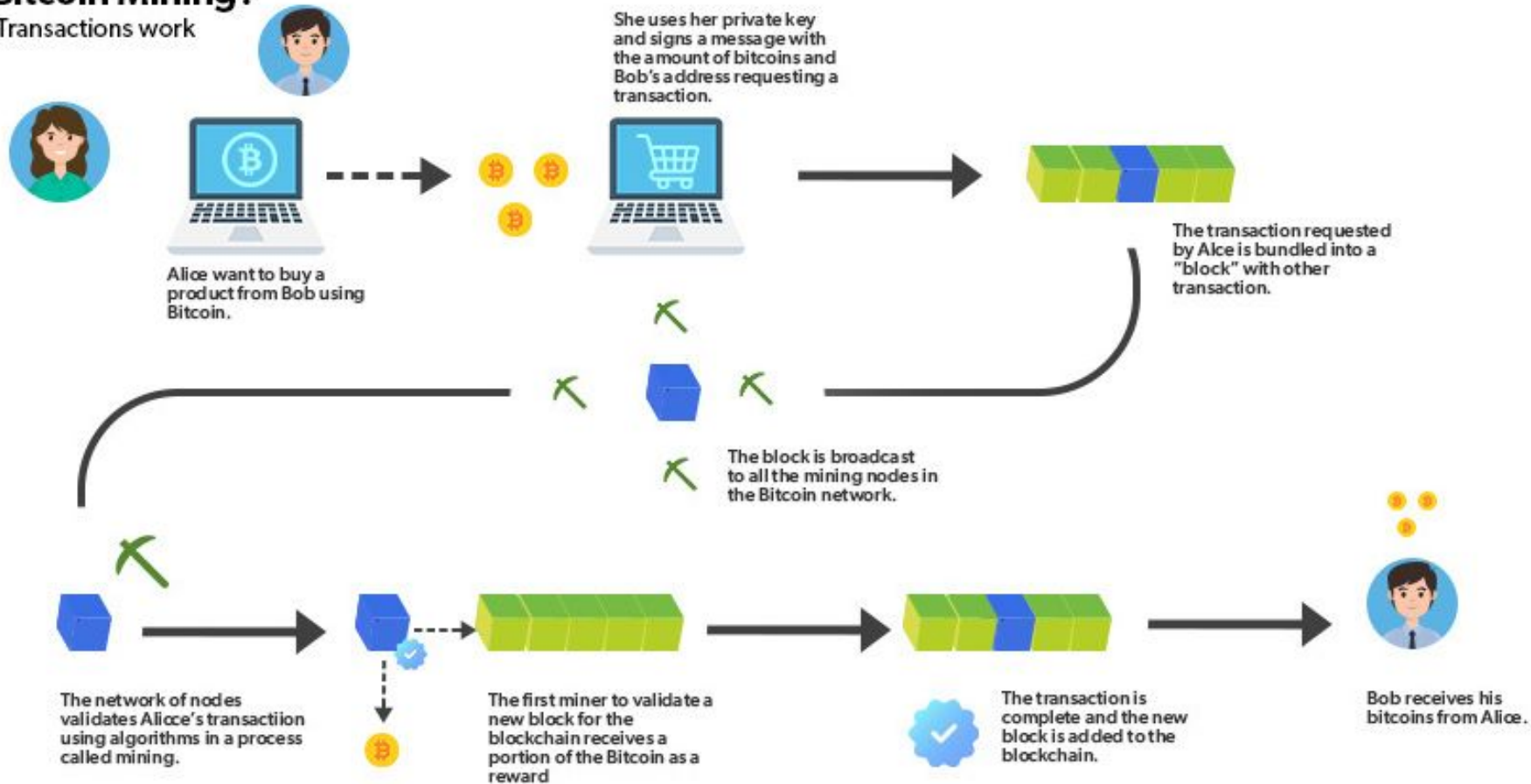
# Comprendre l'EVM et les dApps

# 1.1 Récapitulatif

## Conférence 1

# What is Bitcoin Mining?

How Bitcoin Transactions work

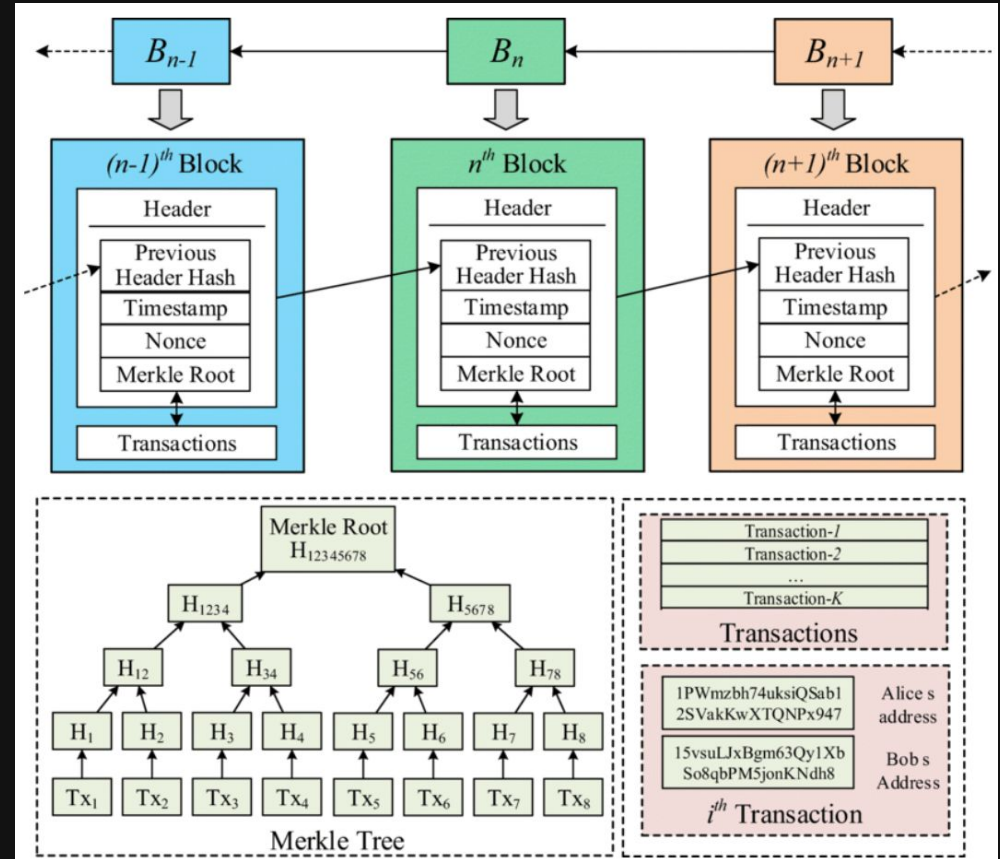


[Source](#)

# Anatomie d'un bloc (1/2)

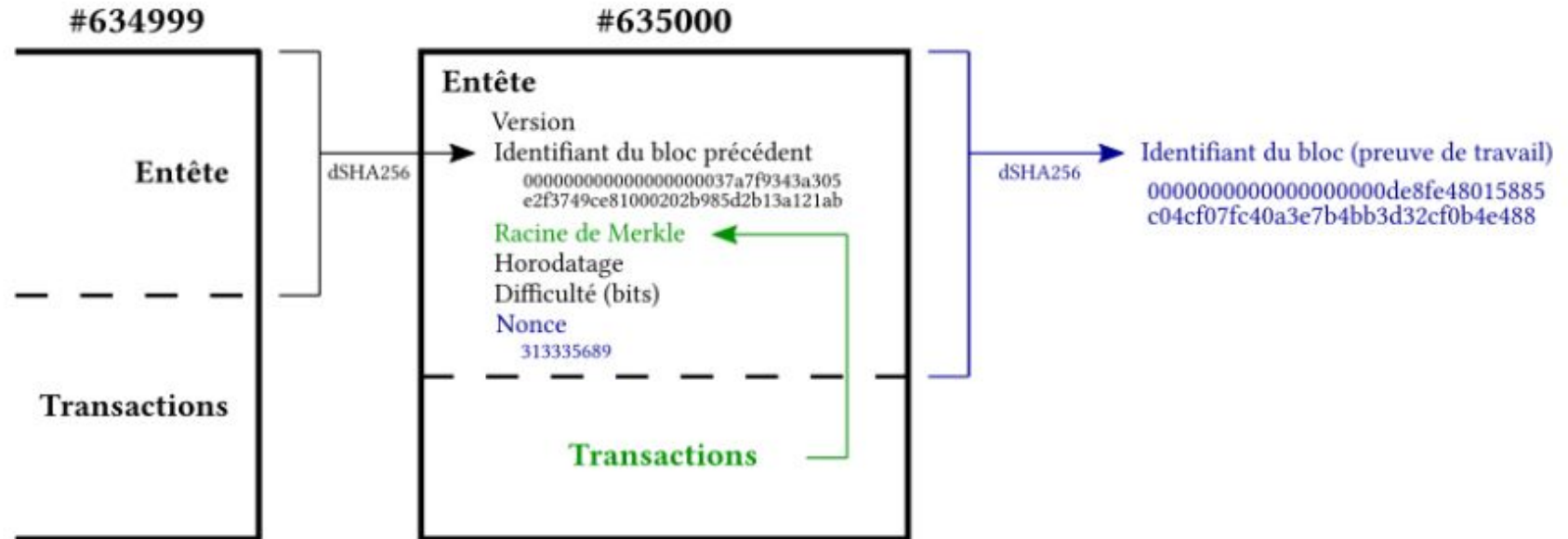
Un bloc est un conteneur numérique qui regroupe les transactions d'une période donnée, avec des éléments comme la racine de Merkle, un timestamp, et la preuve de travail (nonce).

La diffusion et vérification des blocs par le réseau assurent la sécurité et l'intégrité de la blockchain.

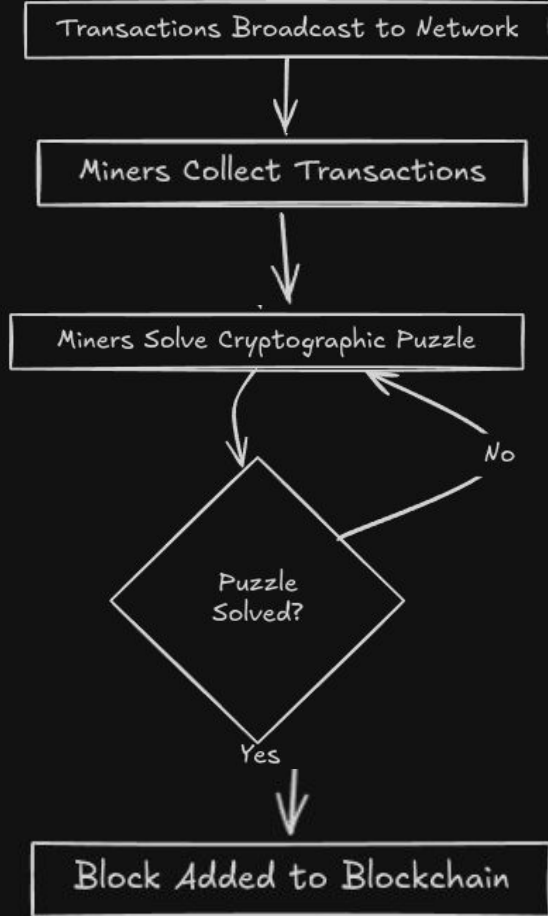


# Proof of Work en Action

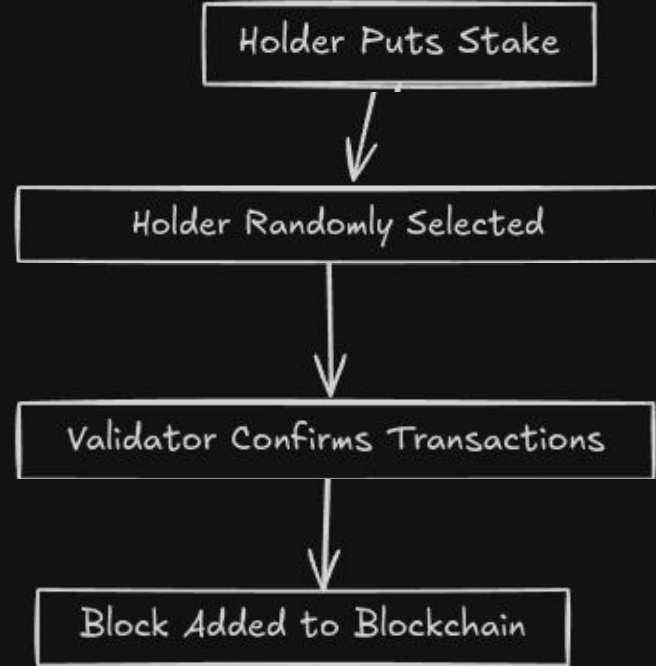
Processus de minage : Les mineurs testent des nonces pour générer un hash respectant une condition (ex. débiter par des zéros). Cette tâche, coûteuse en calcul, est simple à vérifier. Le réseau ajuste dynamiquement la difficulté pour maintenir un intervalle constant entre chaque bloc (par exemple, 10mn pour le Bitcoin).



## Proof of Work

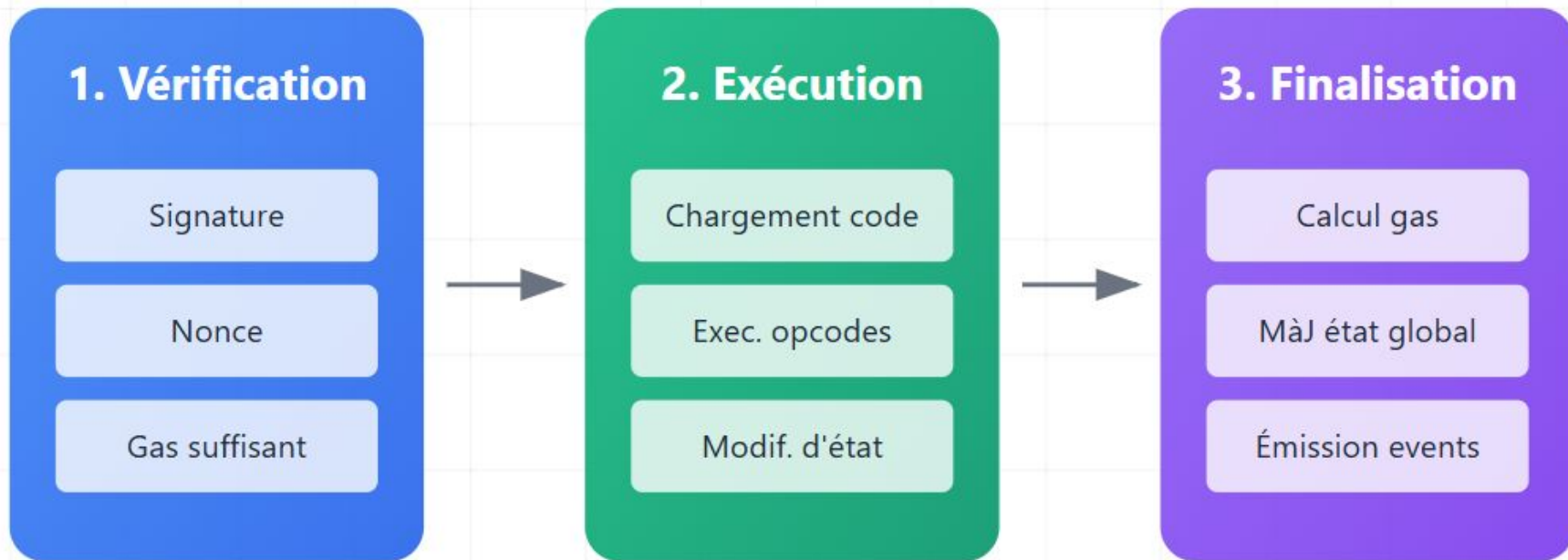


## Proof of Stake



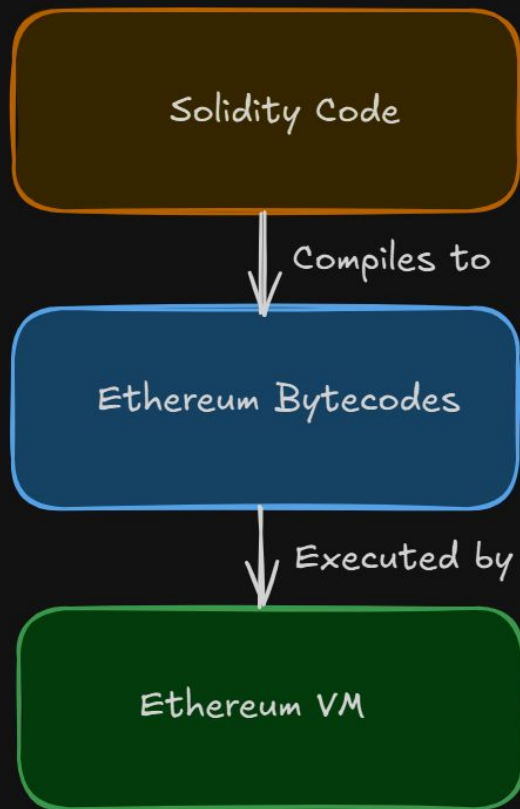
## 2. Architecture de l'EVM

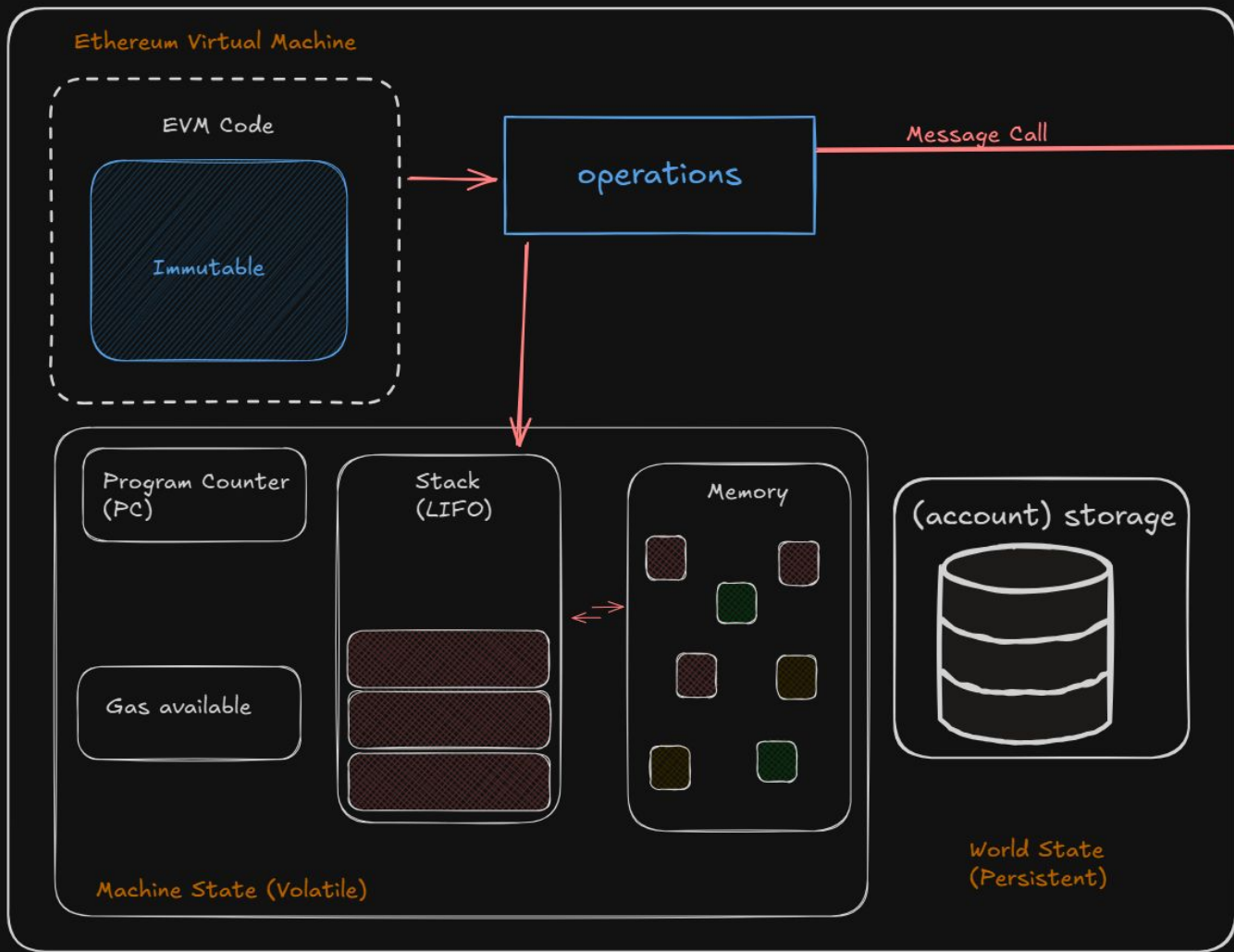
# Lifecycle of a transaction





# From Code to OPCODEs





# Interopérabilité de l'EVM

## Compatible EVM

- Supporte l'exécution de base des smart contracts solidity
- Peut avoir des modifications ou des limitations
- Pas une implémentation exacte de la Machine Virtuelle Ethereum
- Optimisations potentielles des performances ou différences

## Équivalent EVM

- Reproduction fidèle de l'EVM
- Comportement identique pour tous les opcodes et l'exécution
- Totalement interopérable avec l'écosystème Ethereum
- Aucune altération des mécaniques de base de l'EVM

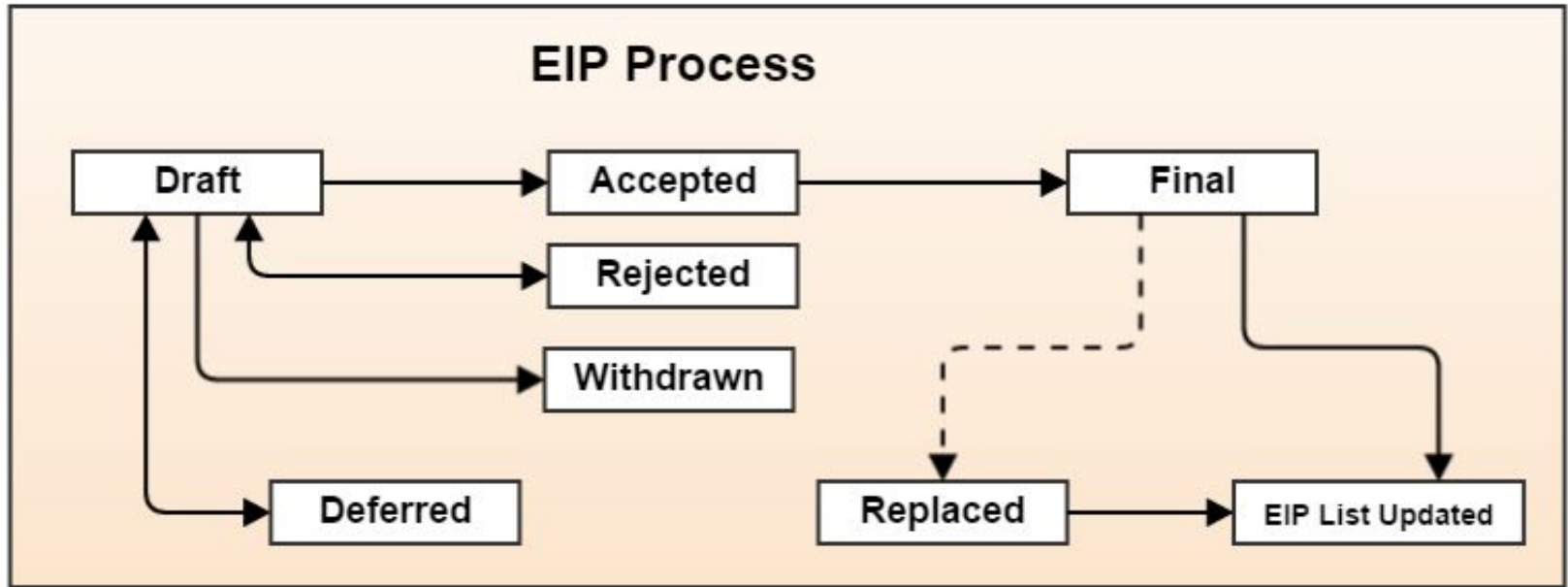
### 3. Fondamentaux Smart Contracts

# Exemple de smart contract basic

```
contract Token {  
    // État  
    mapping(address => uint256) public balances;  
  
    // Événements  
    event Transfer(address from, address to, uint256 amount);  
  
    // Fonctions  
    function transfer(address to, uint256 amount) public {  
        require(balances[msg.sender] >= amount, "Insufficient balance");  
        balances[msg.sender] -= amount;  
        balances[to] += amount;  
        emit Transfer(msg.sender, to, amount);  
    }  
}
```

## 3.1 Standards - EIP / ERC

# Ethereum Improvement Proposal (EIP)



# Ethereum Request for Comment (ERC)

Sous-catégorie d'EIPs, centrée sur les standards au niveau des applications.

Utilisé pour définir des règles d'implémentation pour les tokens, wallets, etc.

Le EIP-20 est devenu l'ERC-20, qui définit les tokens.

Le EIP-721 est devenu le ERC-721 qui sont les NFTs.



# ERC20

Un token "compliant" doit respecter les standards ERC-20:

Implémentation obligatoire des prototypes de fonctions.

Gestionnaires d'événements:  
Transfer, Approval.

Garantit compatibilité et interopérabilité dans l'écosystème.

Exemples connus:

- USDT/USDC (stablecoins)
- Wrapped BTC (WBTC)
- LINK (chainlink)
- UNI (uniswap)
- PEPE, BONK etc

## Fonctions:

```
function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (uint256 balance)
function transfer(address _to, uint256 _value) public returns (bool success)
function transferFrom(address _from, address _to, uint256 _value) public returns (bool success)
function approve(address _spender, uint256 _value) public returns (bool success)
function allowance(address _owner, address _spender) public view returns (uint256 remaining)
```

## Events:

```
event Transfer(address indexed _from, address indexed _to, uint256 _value)
event Approval(address indexed _owner, address indexed _spender, uint256 _value)
```

## States:

```
mapping(address account => uint256) private _balances;
mapping(address account => mapping(address spender => uint256)) private _allowances;
uint256 private _totalSupply;
string private _name;
string private _symbol;
```

# ERC721

L'ERC-721 permet de gérer des tokens non-fongibles (NFT):

**Fonctions clés:** transferts sécurisés, approbations multiples, et gestion des opérateurs

**Événements:** suivis précis des transferts et approbations (Transfer, Approval, ApprovalForAll).

**États :** chaque token a un propriétaire unique, et les données de propriété sont soigneusement mappées. Idéal pour représenter des actifs uniques et indivisibles.

Exemples: Bored Ape Yacht Club (BYAC), CryptoPunk

## Fonctions:

```
function safeTransferFrom(address _from, address _to, uint256 _tokenId, bytes data) external payable;
function safeTransferFrom(address _from, address _to, uint256 _tokenId) external payable;
function setApprovalForAll(address _operator, bool _approved) external;
function getApproved(uint256 _tokenId) external view returns (address);
function isApprovedForAll(address _owner, address _operator) external view returns (bool);
```

## Events:

```
event Transfer(address indexed _from, address indexed _to, uint256 indexed _tokenId);
event Approval(address indexed _owner, address indexed _approved, uint256 indexed _tokenId);
event ApprovalForAll(address indexed _owner, address indexed _operator, bool _approved);
```

## States:

```
string private _name;
// Token symbol
string private _symbol;
mapping(uint256 tokenId => address) private _owners;
mapping(address owner => uint256) private _balances;
mapping(uint256 tokenId => address) private _tokenApprovals;
mapping(address owner => mapping(address operator => bool)) private _operatorApprovals;
```

## 3.2 Patterns

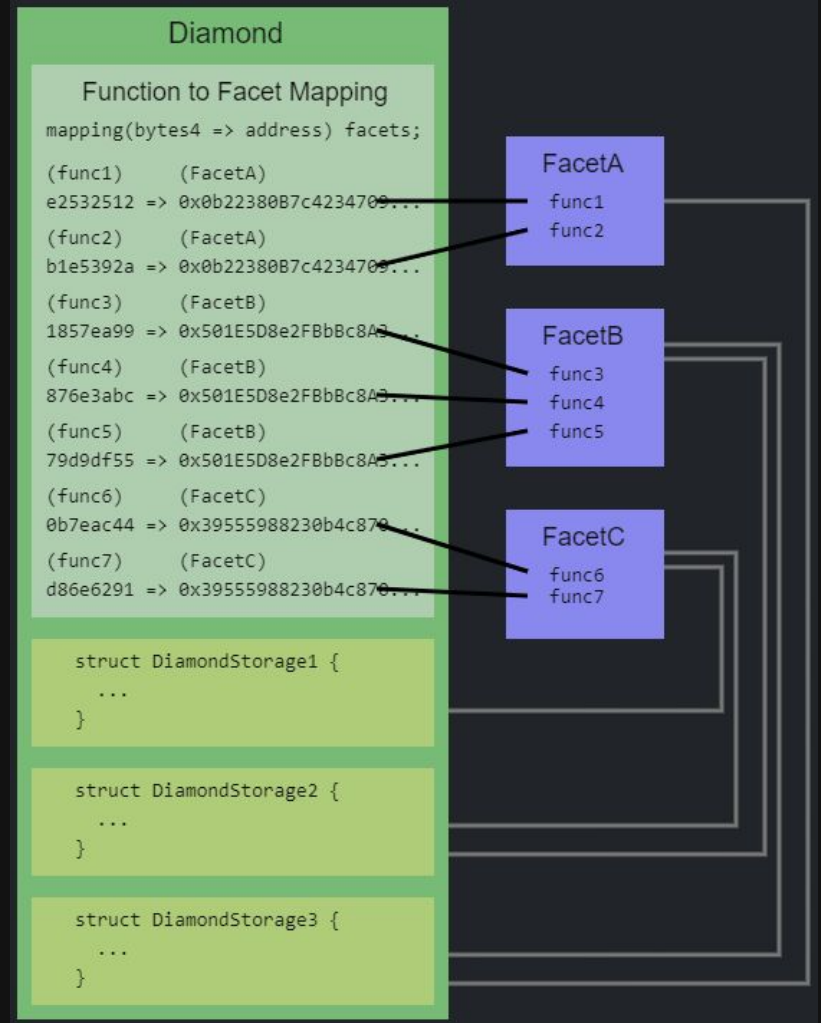
# Diamond Pattern - EIP 2535

Le diamond pattern agit comme un gestionnaire central.

Il contient le state global et les adresses des facets.

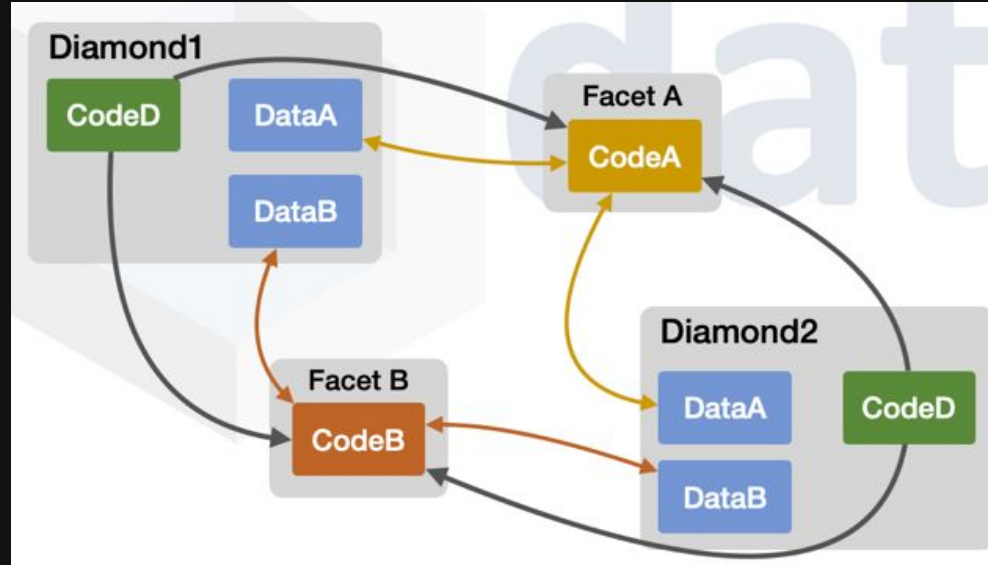
Les facets sont comme des modules qui implémentent des morceaux spécifiques de la logique.

Ce sont des smart contracts a part entière, dont l'adresse est stockée dans le diamond.



# Pourquoi c'est intéressant?

- Les contrats ont une taille limitée ; ce pattern contourne cette limite en déléguant la logique à plusieurs contrats.
- Réutilisation facile de composants pour divers projets, réduisant le travail redondant.
- Sécurité renforcée grâce à l'intégration de modules déjà audités pour des parties critiques.
- Permet des mises à jour modulaires, évitant de redéployer l'ensemble du système.



# Factory Pattern -

C'est un contrat dont le but est de générer dynamiquement plusieurs instances d'un même contrat ou objet.

Par exemple pour encapsuler la création de collections de NFTs



```
// Pattern Factory
contract NFTFactory {
    function createNFT() public returns (address)
    {
        return address(new NFT(msg.sender));
    }
}
```

## 3.3 Sécurité

# Access Control

```
contract AccessControl {
    address public owner;
    mapping(address => bool) public admins;

    modifier onlyOwner() {
        require(msg.sender == owner, "Seul le propriétaire");
        _;
    }

    modifier onlyAdmin() {
        require(admins[msg.sender], "Seul un admin");
        _;
    }

    constructor() {
        owner = msg.sender;
    }

    function addAdmin(address _admin) public onlyOwner {
        admins[_admin] = true;
    }

    function ownerFunction() public onlyOwner {
        // Actions réservées au propriétaire
    }

    function adminFunction() public onlyAdmin {
        // Actions réservées aux admins
    }
}
```



# Reentrancy - Exemple

```
contract VulnerableContract {
    mapping(address => uint) public balances;

    // Deposit funds into the contract
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Withdraw funds from the contract
    function withdraw(uint _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient
balance");

        // Start of reentrancy vulnerability
        (bool success, ) = msg.sender.call{value: _amount}(""); //
External call
        require(success, "Transfer failed");

        balances[msg.sender] -= _amount; // Update the balance after
the call
    }
}
```

```
contract Attack {
    VulnerableContract public vulnerable;

    constructor(address _vulnerableAddress) {
        vulnerable = VulnerableContract(_vulnerableAddress);
    }

    // Fallback function to trigger reentrancy attack
    receive() external payable {
        if (address(vulnerable).balance >= 1 ether) {
            vulnerable.withdraw(1 ether); // Re-enter the vulnerable
contract
        }
    }

    // Initiate the attack
    function attack() public payable {
        require(msg.value >= 1 ether, "Send at least 1 ether to
attack");
        vulnerable.deposit{value: 1 ether}(); // Deposit funds into
the vulnerable contract
        vulnerable.withdraw(1 ether); // Start the withdrawal and
trigger reentrancy
    }
}
```

# Reentrancy - Solution

```
contract FixedContract {
    mapping(address => uint) public balances;

    // Deposit funds into the contract
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    // Withdraw funds from the contract with reentrancy protection
    function withdraw(uint _amount) public {
        require(balances[msg.sender] >= _amount, "Insufficient
balance");

        // First, update the balance before making an external call
        balances[msg.sender] -= _amount;

        // Now make the external call
        (bool success, ) = msg.sender.call{value: _amount}(""); //
External call
        require(success, "Transfer failed");
    }
}
```

# MEV - Front-Running en Blockchain

## Définition:

Le **front-running** se produit lorsqu'un acteur malveillant (le "frontrunner") exploite la visibilité des transactions non confirmées dans un mempool pour en soumettre une transaction avant celle d'un utilisateur ciblé, dans le but de bénéficier de l'exécution anticipée.

## Conséquences :

- Perte de valeur pour l'utilisateur victime.
- Impact négatif sur la confiance dans les échanges.

## Comment le prévenir ?

- **RPC dédié et solutions de batchs :** limite les opportunités de front-running.
- **Optimisation des frais de gas :** Pour éviter les attaques basées sur les frais.

## 4. les dApps

## 4.1 Architecture

### Frontend - Smart Contracts

# Connection avec la Blockchain

- Un provider, qui est une librairie agissant comme interface pour la blockchain (ex. ethers.providers.Web3Provider).
- Un wallet comme MetaMask (window.ethereum) pour signer les transactions et accéder aux comptes utilisateur.
- RPC (Remote Procedure Call) : remplace HTTP pour envoyer des requêtes et recevoir des réponses depuis la blockchain. Il permet de se connecter à un réseau blockchain via un service comme Infura ou Alchemy, sans avoir besoin d'un nœud local.

```
const setupWeb3 = async () => {  
  if (!window.ethereum) {  
    throw new Error("Please install MetaMask!");  
  }  
  
  const provider = new ethers.providers.Web3Provider(window.ethereum);  
  await provider.send("eth_requestAccounts", []);  
  return provider.getSigner();  
};
```

# Interaction avec un smart contract

Une dApp combine un frontend traditionnel et des smart contracts déployés sur la blockchain.

Grâce à des bibliothèques comme ethers.js, on peut :

- Instancier un contrat via son adresse et son ABI\*.
- Appeler des fonctions du contrat (ex. transfer) et gérer les transactions en temps réel

*Application Binary Interface (ABI)*: Dans ce contexte, une liste des prototypes des fonctions d'un smart contract, comme un \*.h en C

```
const TokenContract = ({ address, abi, signer }) => {  
  const contract = new ethers.Contract(address, abi, signer);  
  
  async function transfer(to, amount) {  
    try {  
      const tx = await contract.transfer(to, amount);  
      await tx.wait();  
      return true;  
    } catch (error) {  
      console.error(error);  
      return false;  
    }  
  }  
  
  return { transfer };  
};
```

## 4.2 Gestion des Events & Etats




# Suivi des Événements Blockchain

**Écoute d'événements** : Un smart contract peut émettre des événements que vous pouvez "écouter" depuis le frontend.

**Réactivité en temps réel** : Ce mécanisme permet d'agir instantanément en réponse à un changement d'état sur la blockchain.

**Exemple** : "Transfert", permet de suivre des transferts de tokens ou d'autres actions importantes dans le contrat.



```
contract.on("Transfer", (from, to, amount, event) => {  
    console.log(`Transfer: ${from} → ${to}: ${amount}`);  
    updateUI();  
});
```

# Suivis des states on-chain

## États/States:

**idle**: État initial, aucune transaction en cours

**waiting\_user**: En attente de confirmation utilisateur dans MetaMask

**pending**: Transaction envoyée, en attente de confirmation blockchain

**success**: Transaction confirmée avec succès

**error**: Échec de la transaction

```
const [txState, setTxState] = useState('idle');

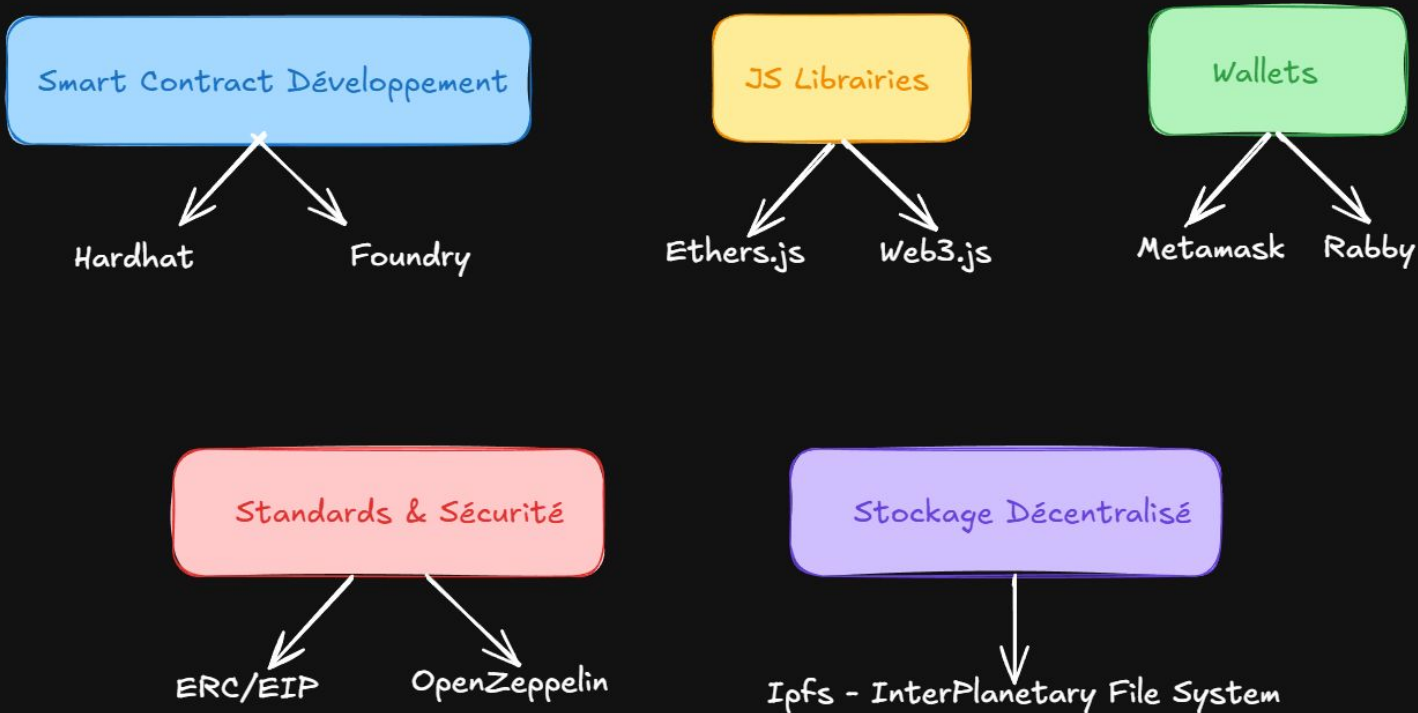
const sendTransaction = async () => {
  try {
    setTxState('waiting_user');
    const tx = await contract.transfer(to, amount);

    setTxState('pending');
    await tx.wait(); // Attente confirmation blockchain

    setTxState('success');
  } catch (error) {
    setTxState('error');
    console.error(error);
  }
};

// Affichage UI conditionnel
const TransactionStatus = () => {
  switch(txState) {
    case 'waiting_user':
      return <div>Confirmez la transaction dans MetaMask...</div>;
    case 'pending':
      return <div>Transaction en cours...</div>;
    case 'success':
      return <div>Transaction réussie!</div>;
    case 'error':
      return <div>Erreur lors de la transaction</div>;
    default:
      return null;
  }
};
```

## 4.3 Outils et framework de développement



## 4.4 Live dApps Demos

# Uniswap - AMM (Automated Market Maker)

## Qu'est-ce qu'un AMM ?

Un **AMM** est un type de protocole utilisé dans les échanges décentralisés (DEX) qui permet de négocier des actifs sans avoir besoin d'un carnet d'ordres centralisé. L'AMM utilise des **smart contracts** pour établir des prix et exécuter des échanges.

## Uniswap : Le Leader du Marché

- **Uniswap** présent sur de multiples EVM chains.
- Il permet aux utilisateurs de fournir des liquidités à un pool, et en retour, ils reçoivent des **frais de transaction** générés par les échanges dans ce pool.

## Comment fonctionne Uniswap ?

- **Les pools de liquidités** : Les utilisateurs déposent une paire d'actifs (par exemple, ETH et DAI) dans un smart contract.
- **La formule de l'AMM** : Uniswap utilise une formule de produit constant :  $x \times y = k$  ou  $x \times y = k$ , où  $x$  et  $y$  sont les quantités des deux actifs dans le pool, et  $k$  est une constante.
- **Prix dynamique** : Le prix des tokens dans le pool est ajusté automatiquement en fonction de l'offre et de la demande, sans carnet d'ordres.

# Uniswap - AMM (Automated Market Maker)

```
// Exemple simplifié du contrat Uniswap v2 AMM
contract UniswapV2AMM {
    mapping(address => uint) public liquidity;
    mapping(address => uint) public tokenBalances;

    // Déposer des liquidités dans le pool
    function provideLiquidity(address tokenA, uint amountA, address tokenB, uint amountB) public {
        tokenBalances[tokenA] += amountA;
        tokenBalances[tokenB] += amountB;
        liquidity[msg.sender] += amountA + amountB;
    }

    // Échanger des tokens
    function swap(address tokenIn, uint amountIn, address tokenOut) public {
        uint amountOut = getAmountOut(amountIn, tokenIn, tokenOut);
        tokenBalances[tokenIn] -= amountIn;
        tokenBalances[tokenOut] += amountOut;
    }

    // Calcul du prix basé sur la formule  $x * y = k$ 
    function getAmountOut(uint amountIn, address tokenIn, address tokenOut) public view returns (uint) {
        uint reserveIn = tokenBalances[tokenIn];
        uint reserveOut = tokenBalances[tokenOut];
        return (amountIn * reserveOut) / reserveIn;
    }
}
```

# Quack

## Qu'est-ce que le Play-to-Earn ?

Le **Play-to-Earn (P2E)** est un modèle économique où les joueurs peuvent gagner des récompenses en jouant à des jeux vidéo basés sur la blockchain. Ces récompenses peuvent inclure des tokens, des objets numériques (NFTs), ou des cryptomonnaies.

## Objectifs de Quack - Forbidden Duck of Wisdom:

Faisant suite à notre projet pour **42 Lausanne**, Cod'Hash travail sur un écosystème de multiples jeux **Play-to-Earn** où les **joueurs** peuvent **créer des items** (NFTs), et les **studios** leur propre **Game Mode** de manière permissionless.

## Création d'Items Permissionless :

- **Propriétés Décentralisées** : Les joueurs peuvent créer, échanger et vendre leurs items sans avoir besoin de permission d'une entité centrale.
- **NFTs** : Chaque item créé dans le jeu est un **NFT** unique qui peut être échangé sur des marketplaces comme TokenFabric.xyz ou OpenSea.
- **Mécanisme de Gouvernance** : Les décisions concernant le jeu, et l'évolution du système peuvent être prises par la communauté via un DAO (Decentralized Autonomous Organization).

