To write and deploy smart contracts for different consensus mechanisms (PoOW, zkEVM, and PoE) on the Polygon platform, follow these steps:

1. Setting Up the Development Environment

1. Install Prerequisites:

- o Install **Node.js** and **npm** (Node Package Manager).
- o Install the **Truffle** or **Hardhat** framework for writing and testing smart contracts.
- o Install the **MetaMask** wallet to interact with the blockchain.
- o Install the **Polygon CLI** (optional for Polygon-specific tools).

2. Choose an IDE:

 Use an IDE like Visual Studio Code with the Solidity extension for writing smart contracts.

3. Set Up Polygon Testnet:

- Configure MetaMask to connect to Polygon's Mumbai Testnet for testing. Add the RPC URL and chain ID to your wallet.
- o Obtain test MATIC from the Polygon Faucet for deploying contracts.

2. Writing Smart Contracts

(a) Smart Contract for Proof of Ownership without ZKP (PoOW)

1. Functionality:

- o Store the hash of an image.
- o Allow users to claim ownership of a file by verifying the hash.
- Store metadata like timestamps and owner addresses.

2. Code Example (Solidity):

```
pragma solidity ^0.8.0;

contract ProofOfOwnership {

struct File {

address owner;

string fileHash;

uint256 timestamp;
```

}

```
mapping(string => File) public files;
  function registerFile(string memory fileHash) public {
    require(files[fileHash].owner == address(0), "File already exists");
    files[fileHash] = File(msg.sender, fileHash, block.timestamp);
  }
  function getFile(string memory fileHash) public view returns (address, uint256) {
    require(files[fileHash].owner != address(0), "File not registered");
    return (files[fileHash].owner, files[fileHash].timestamp);
  }
}
(b) Smart Contract for zkEVM
    1. Functionality:
            o Use zk-proofs to verify ownership or validity without revealing sensitive data.
                Requires integration with a zkEVM-compatible library like Circom or Snark.js.
    2. Code Example:
pragma solidity ^0.8.0;
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
contract zkEVM {
  using ECDSA for bytes32;
  struct Proof {
    bytes32 proofHash;
    address prover;
    uint256 timestamp;
  }
```

```
mapping(bytes32 => Proof) public proofs;

function submitProof(bytes32 proofHash) public {
    require(proofs[proofHash].prover == address(0), "Proof already submitted");
    proofs[proofHash] = Proof(proofHash, msg.sender, block.timestamp);
}

function verifyProof(bytes32 proofHash) public view returns (address, uint256) {
    require(proofs[proofHash].prover != address(0), "Proof not found");
    return (proofs[proofHash].prover, proofs[proofHash].timestamp);
}
```

For full zkEVM implementation, off-chain zk circuits and proof generation using **Circom** or **zk-SNARK tools** are required.

Smart Contract for Proof of Existence (PoE)

1. Functionality:

- o Store a hash of the image to prove its existence at a certain timestamp.
- o This is simpler and lightweight compared to PoOW or zkEVM.

2. Code Example:

```
pragma solidity ^0.8.0;

contract ProofOfExistence {
   mapping(bytes32 => uint256) public timestamps;

function registerDocument(bytes32 documentHash) public {
   require(timestamps[documentHash] == 0, "Document already exists");
   timestamps[documentHash] = block.timestamp;
  }

function getTimestamp(bytes32 documentHash) public view returns (uint256) {
   require(timestamps[documentHash] != 0, "Document not registered");
```

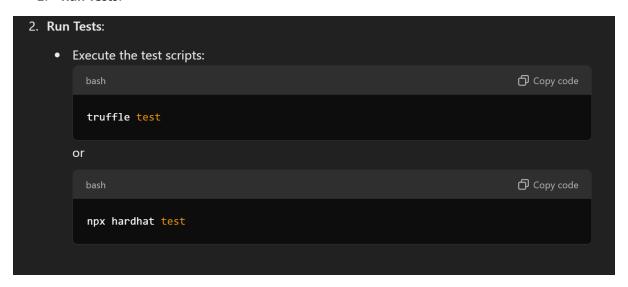
```
return timestamps[documentHash];
}
```

3. Testing the Smart Contracts

1. Write Test Cases:

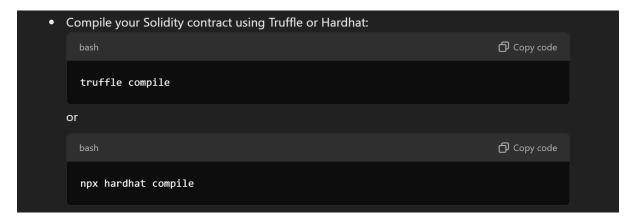
- Use frameworks like Truffle or Hardhat to write and execute tests.
- Simulate transactions to check functionality (e.g., registering files, verifying ownership).

2. Run Tests:



Deploying the Smart Contracts on Polygon

1. Compile the Contract:



Deploy to Mumbai Testnet:

• Update the deployment script to specify the Polygon Mumbai Testnet RPC URL and your wallet's private key.



1. Verify Deployment:

 Use a blockchain explorer like **Polygonscan** to verify your smart contract address and deployment.

5. Interacting with the Deployed Contracts

1. Frontend Integration:

- Use web3.js or ethers.js in your frontend application to interact with the deployed smart contracts.
- o For example:

```
javascript

const contract = new web3.eth.Contract(ABI, contractAddress);
await contract.methods.registerFile(fileHash).send({ from: userAddress });
```

const contract = new web3.eth.Contract(ABI, contractAddress);

await contract.methods.registerFile(fileHash).send({ from: userAddress });

1. Transaction Testing:

 Test the performance of each consensus mechanism by executing transactions and measuring metrics like latency, cost, and throughput.

6. Results and Analysis

- 1. Measure performance metrics for each consensus mechanism:
 - PoOW: Evaluate basic ownership claims with no cryptographic overhead.
 - o **zkEVM**: Assess latency and computational performance due to zk-proof generation.
 - o **PoE**: Measure lightweight execution speed and energy efficiency.

2.	Compare the results across the parameters defined: computational performance, latency, throughput, energy efficiency, cost efficiency, fault tolerance .