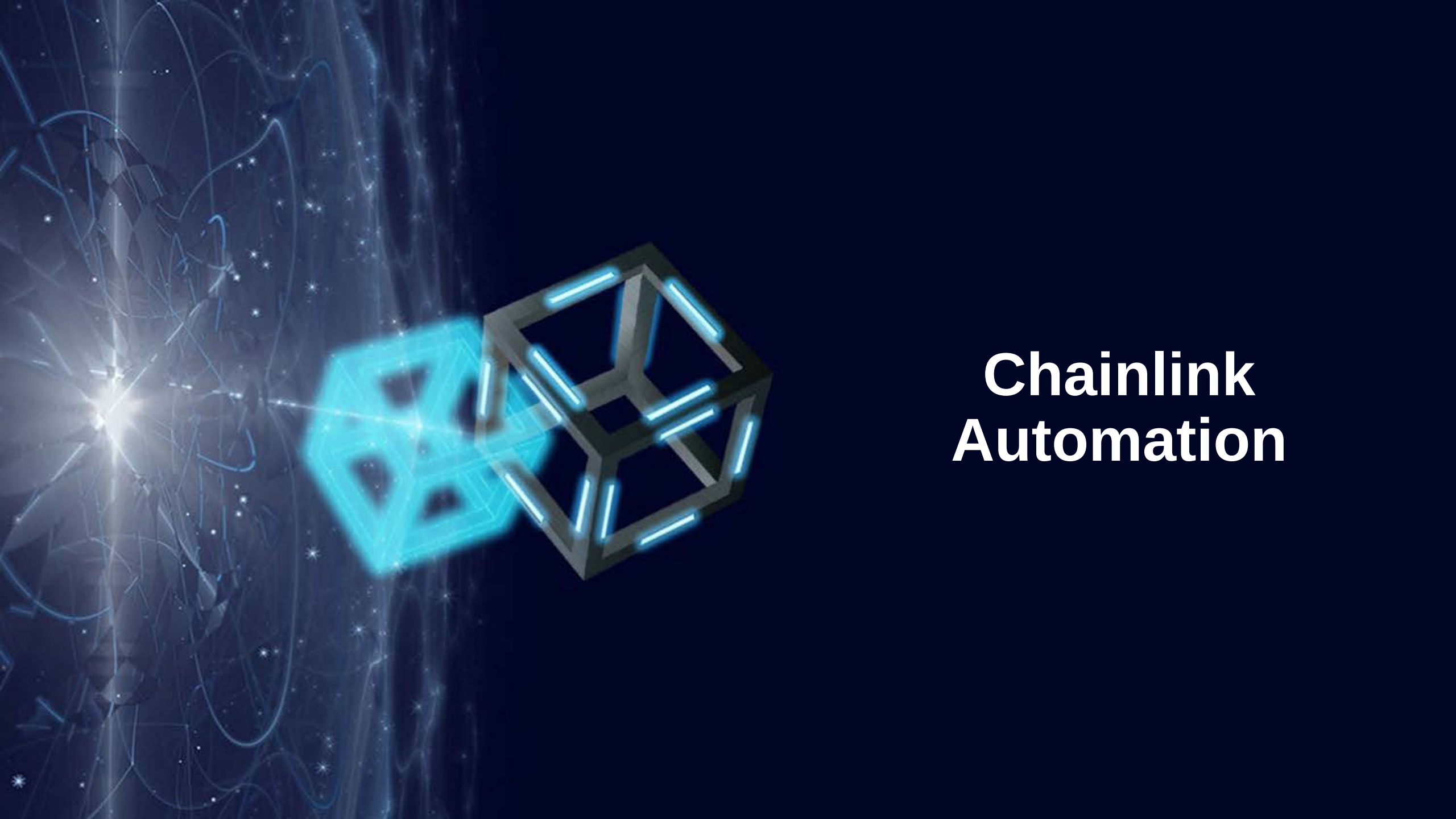
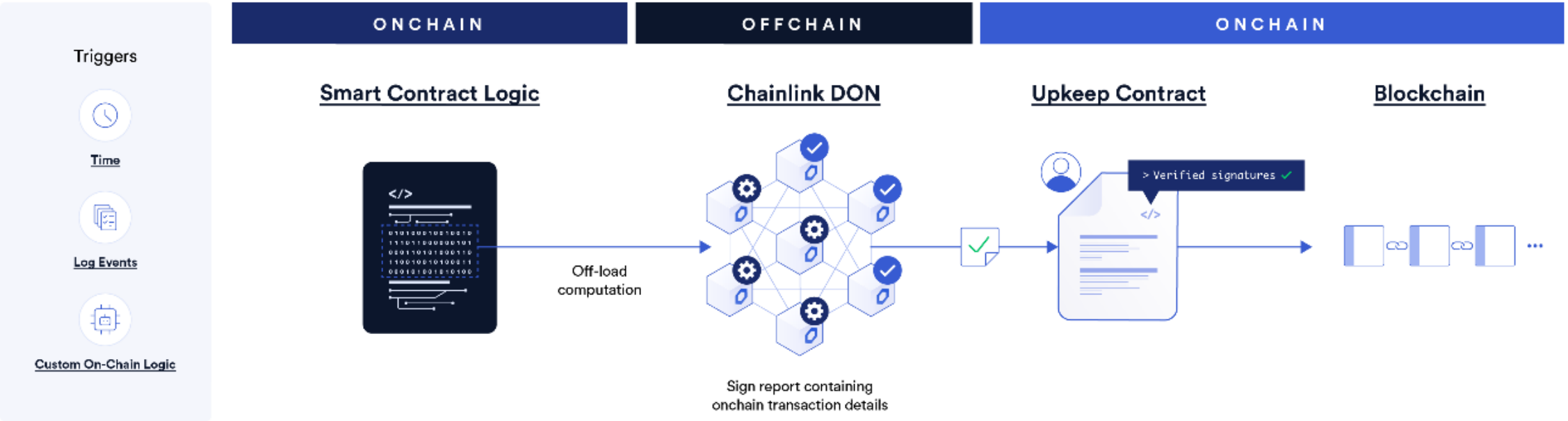

ETHEREUM SMART
CONTRACT DEVELOPMENT
Chainlink Automation
Upgradeable Contracts - UUPS



# Chainlink Automation

# Chainlink Automation

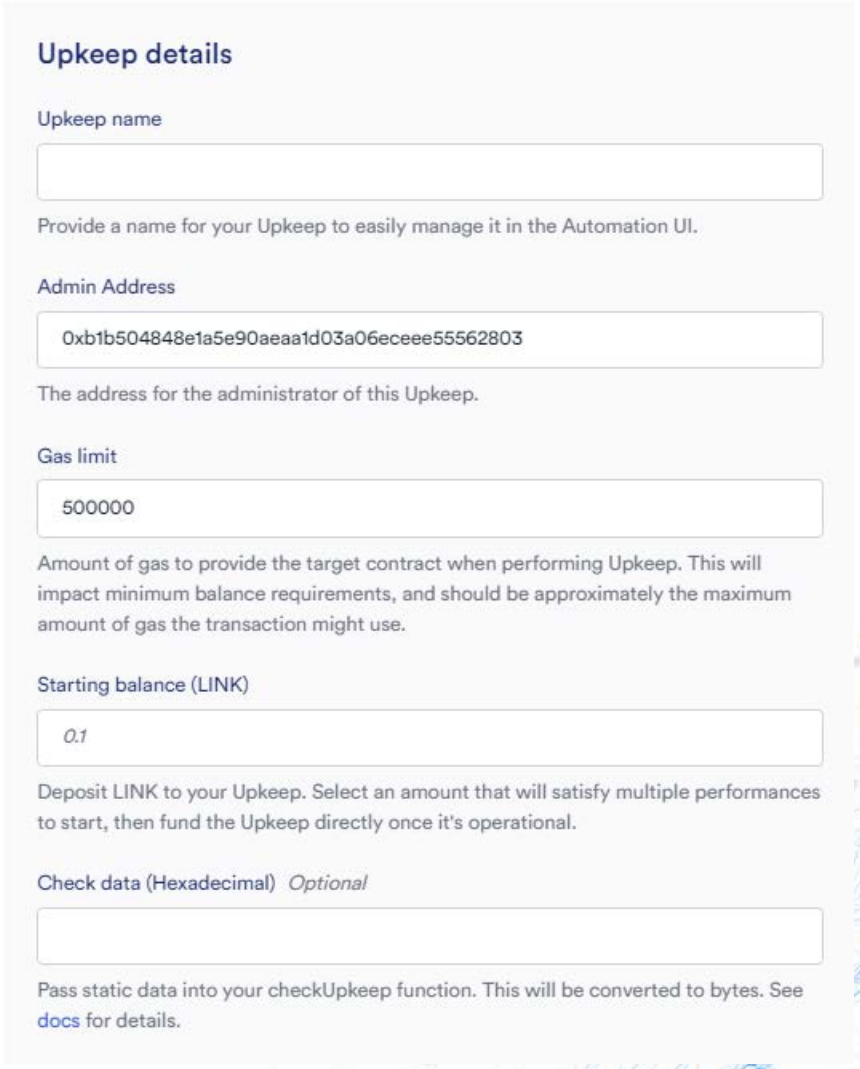
- Chainlink automation executes smart contract functions using a variety of triggers: Time-based, custom logic (evaluated offchain) & log triggered
- Automation nodes simulate "***checkUpkeep***" functions => determine when upkeeps need to be performed
- The signed report contains the "***performData***" that will be executed onchain



# Registering an Upkeep

- Select a network and register an Upkeep: https://automation.chain.link

- Choose a trigger, provide a target contract address and upkeep details : upkeep name, admin address, gas limit, LINK balance, data provided for the "***checkUpkeep***" function (optional)



# Test LINK & Token Contracts

- Get test LINK : https://faucets.chain.link/sepolia
- LINK token contracts for mainnet, Sepolia, BNB, Avalanche, Polygon, Arbitrum, Optimism... : https://docs.chain.link/resources/link-token-contracts
- Sepolia LINK address : 0x779877A7B0D9E8603169DdbD7836e478b4624789

# Automation Compatible Contracts

**The interface your contract needs to implement depends on the trigger you want to use:**

- For a time-based trigger, no interface is required
- For a custom-logic trigger, the ***AutomationCompatibleInterface*** interface is required
- For a log trigger, the ***ILogAutomation*** interface is required

# Requirements for Custom Logic Compatible Contracts

- Import ***AutomationCompatibleInterface*** from AutomationCompatibleInterface.sol

- Include a ***checkUpkeep*** function => contains the logic that will be executed offchain to see if ***performUpkeep*** should be executed. checkUpkeep can use onchain data and a checkData parameter (bytes) to perform complex calculations offchain and then send the result to performUpkeep as ***bytes performData***

  ***function checkUpkeep(bytes calldata checkData) external view override returns (bool upkeepNeeded, bytes memory performData)***

- Include a ***performUpkeep*** function that will be executed onchain when ***checkUpkeep*** returns true

  ***function performUpkeep(bytes calldata performData) external***

- Once the upkeep is registered, the Chainlink Automation Network frequently simulates the checkUpkeep function offchain to determine if performUpkeep needs to be called

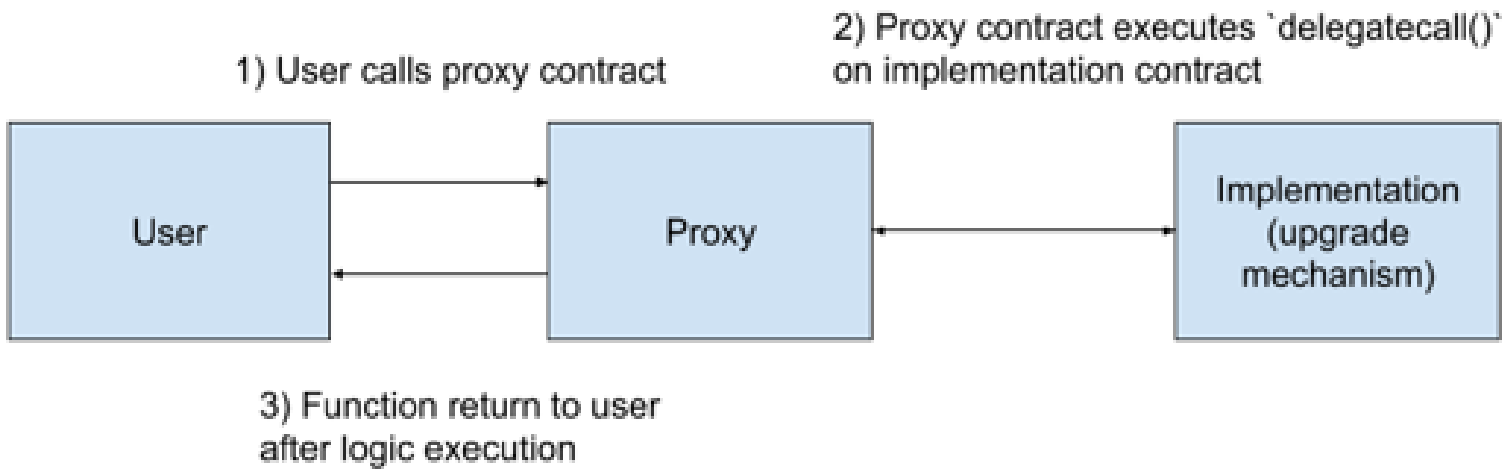
- When checkUpkeep returns true, the Chainlink Automation Network calls performUpkeep onchain

# Perform Complex Computations With No Gas Fees

- Add gas intensive computations to the checkUpkeep() function
- This computation doesn’t consume any gas - Automation Nodes perform the computation offchain
- Multiple upkeeps van be used for the same contract to do the work in parallel

# Upgradeable Contracts - UUPS

# Upgradeable Contracts




- The client always interacts with the proxy contract
- The proxy performs a ***delegatecall()*** on the implementation contract
- The implementation contract modifies the storage of the proxy contract

# UUPS - Universal Upgradeable Proxy Standard

- The upgrade is triggered via the logic contract
- There is a unique storage slot in the proxy contract to store the address of the logic contract that it points to
- Whenever the logic contract is upgraded, that storage slot is updated with the new logic contract address

# Initializing an Implementation Contract

- Code inside a constructor is not part of the runtime bytecode => executed only once, during deployment
- The code within a logic contract's constructor will never be executed in the context of the proxy's state
- Solution: move the constructor code to an "***initialize***" function & call it when the proxy links to the logic

**Required packages:**

- npm i @openzeppelin/***contracts-upgradeable*** @openzeppelin/***hardhat-upgrades*** --save-dev

**Key methods:**

- ***initialize()*** : Upgradable contracts should have an initialize method in place of constructors, and also the ***initializer*** modifier to make sure the contract is initialized only once
- ***_authorizeUpgrade()*** : Safeguard from unauthorized upgrades using the ***onlyOwner*** modifier

# Example: Create & Deploy an Upgradeable Contract

- Create a simple upgradeable contract (Pizza) using the UUPS proxy pattern
- Deploy the contract using the ***upgrades.deployProxy()*** function

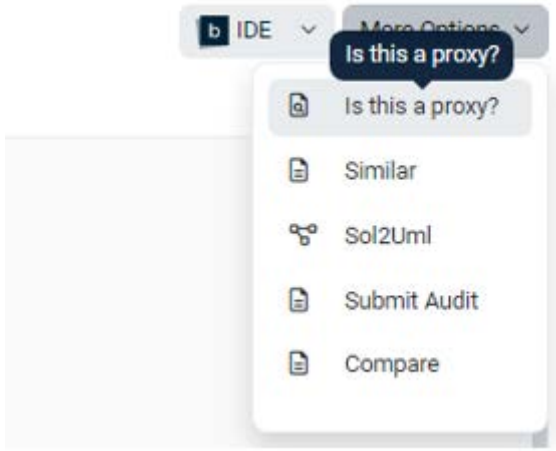
```
3   const PizzaFactory = await ethers.getContractFactory("Pizza")
4
5   const pizza = await upgrades.deployProxy(PizzaFactory, [SLICES], {
6       initializer: "initialize",
7       kind: "uups",
8   })
9
10  await pizza.waitForDeployment()
```

**Two new contracts are deployed via two transactions:**

- the first one is the actual Pizza contract (the implementation contract)
- the second is the proxy contract

# Verifying the Contract

- **npx hardhat verify --network sepolia  CONTRACT_ADDRESS**
- We interact with the Pizza contract via the proxy contract => inform Etherscan that the contract is a proxy
- In the Contract tab of the proxy contract, choose ***“Is this a proxy?”*** from the dropdown and click ***Verify***




- To interact with the contract, use the "Read as Proxy" and "Write as Proxy" options in the Proxy Contract tab

# Upgrading the Contract

- Create a new version of the Pizza contract => add/modify functions
- Upgrade the existing Pizza contract using the ***upgrades.upgradeProxy()*** function

```
3   const PizzaV2Factory = await ethers.getContractFactory("PizzaV2")
4
5   await upgrades.upgradeProxy(PROXY_CONTRACT_ADDRESS, PizzaV2Factory)
6
```

**The upgradeProxy() function creates 2 transactions:**

- Deployment of the PizzaV2 contract
- Call "***upgradeTo***" on PizzaV2 to perform an upgrade => proxy contract points to the newly deployed PizzaV2

# Upgradeable Smart Contract Libraries

- Use the Upgradeable variant of OpenZeppelin Contracts
- Define your own public initializer function and call the parent initializer of the contract you extend
- Naming convention for parent initializer: ***__{ContractName}_init***

```
3  import {ERC721Upgradeable} from "@openzeppelin/contracts-upgradeable/token/ERC721/ERC721Upgradeable.sol";
4
5  contract MyUpgradeableNFT is ERC721Upgradeable {
6      function initialize() public initializer {
7          __ERC721_init("MyNFT", "MNFT");
8      }
9  }
```

# Inheritance and initializers

- Solidity takes care of automatically invoking the constructors of all ancestors of a contract
- In upgradeable contracts, the initializers of all parent contracts need to be called manually
- The initializer modifier can only be called once even when using inheritance => parent contracts should use the "***onlyInitializing***" modifier

```
3   contract BaseContract is Initializable {
4       uint256 public x;
5
6       function initialize() public onlyInitializing {
7           x = 42;
8       }
9   }
10
11  contract MyContract is BaseContract {
12      uint256 public y;
13
14      function initialize(uint256 _y) public initializer {
15          BaseContract.initialize();
16          y = _y;
17      }
18  }
```

# Avoid Initial Values in Field Declarations

- Using initial values is equivalent to setting these values in the constructor
- Initial values must be set in an initializer function otherwise they won't be set
- It is ok to define constant state variables => the compiler does not reserve a storage slot for these variables

```
3   contract MyContract is Initializable {
4       uint256 public hasInitialValue;
5
6       function initialize() public initializer {
7           hasInitialValue = 42;
8       }
9   }
```

# Modifying an Implementation Contract

**Avoid storage collisions between implementation contracts:**

- Don't change the order of state variables
- Don't change the type of state variables
- Don’t introduce new state variable(s) before existing ones
- Add new state variable(s) at the end

```
3   contract MyContractV1 {
4       uint256 private x;
5       string private y;
6   }
7
8   contract MyContractV2 {
9       string private y;
10      uint256 private x;
11  }
```

**We cannot add new variables to base contracts,
if the child contract also has storage variables:**

- If an additional variable is added to the base contract, it will be assigned the slot the child contract had in the previous version

```
3   contract Base {
4       uint256 base1;
5   }
6
7
8   contract Child is Base {
9       uint256 child;
10  }
```

# Storage Gaps

- Allow to reserve storage slots in a base contract => future versions of that contract can use up those slots without affecting the storage layout of child contracts
- To create a storage gap, declare a fixed-size array in the base contract with an initial number of slots.
- Declare an array of uint256 so that each element reserves a 32 byte slot.
- Use the name ***__gap*** or a name starting with ***__gap_*** for the array so that OpenZeppelin Upgrades will recognize the gap
- If the base contract later needs to add extra variable(s), reduce the appropriate number of slots from the storage gap

```
3   contract BaseV1 {
4       uint256 base1;
5       uint256[49] __gap;
6   }
7
8   contract Child is Base {
9       uint256 child;
10  }
11
12  contract BaseV2 {
13      uint256 base1;
14      uint256 base2; // 32 bytes
15      uint256[48] __gap;
16  }
```