Part 1: Question

1. How does permit2 works?

Permit2 is a solution that enhances the usage of ERC20 tokens by enabling more secure and efficient approval and transaction processes.

In terms of architecture, Permit2 consists of two contracts `AllowanceTransfer` and `SignatureTransfer`.

I would like to explain how the system operates:

1. **Permit-Based Approval Mechanism**: Permit2 allows users to pre-approve token transfers without the need for a traditional `approve()` transaction. This is done by signing a message (a "permit") off-chain and later submitting it on-chain using the `permit()` function, along with the necessary parameters.

```
function permit(address owner, PermitSingle memory permitSingle, bytes
calldata signature) external {
    if (block.timestamp > permitSingle.sigDeadline) revert
SignatureExpired(permitSingle.sigDeadline);

    // Verify the signer address from the signature.
    signature.verify(_hashTypedData(permitSingle.hash()), owner);

    _updateApproval(permitSingle.details, owner, permitSingle.spender);
}
```

2. **Batch Permitting**: Permit2 also supports batch permitting, which enables users to sign multiple permits within a single off-chain message. This reduces gas costs and increases efficiency when interacting with multiple tokens or spenders.

```
function permit(address owner, PermitBatch memory permitBatch, bytes
calldata signature) external {
    if (block.timestamp > permitBatch.sigDeadline) revert
SignatureExpired(permitBatch.sigDeadline);

    // Verify the signer address from the signature.
    signature.verify(_hashTypedData(permitBatch.hash()), owner);

    address spender = permitBatch.spender;
    unchecked {
        uint256 length = permitBatch.details.length;
        for (uint256 i = 0; i < length; ++i) {
            _updateApproval(permitBatch.details[i], owner, spender);
        }
    }
}
```

3. **Token Transfers**: The system supports direct token transfers from the `from` address to the `to` address if the spender has sufficient allowance. It also supports batch transfers.

```
function transferFrom(address from, address to, uint160 amount, address
token) external {
    _transfer(from, to, amount, token);
}
```

4. **Nonce Invalidations**: Users can also invalidate their nonces to prevent replay attacks. This is done using the `invalidateNonces()` function, where a user can set a new nonce that is greater than the current one.

```
function invalidateNonces(address token, address spender, uint48 newNonce)
external {
    uint48 oldNonce = allowance[msg.sender][token][spender].nonce;
    if (newNonce <= oldNonce) revert InvalidNonce();

    // Limit the amount of nonces that can be invalidated in one
transaction.
    unchecked {
        uint48 delta = newNonce - oldNonce;
        if (delta > type(uint16).max) revert ExcessiveInvalidation();
    }

    allowance[msg.sender][token][spender].nonce = newNonce;
    emit NonceInvalidation(msg.sender, token, spender, newNonce, oldNonce);
}
```

5. **Lockdown Mechanism**: The contract also provides a lockdown mechanism. By calling `lockdown()` function, a user can immediately revoke the allowances of multiple tokens for multiple spenders.

```
function lockdown(TokenSpenderPair[] calldata approvals) external {
    address owner = msg.sender;
    // Revoke allowances for each pair of spenders and tokens.
    unchecked {
        uint256 length = approvals.length;
        for (uint256 i = 0; i < length; ++i) {
            address token = approvals[i].token;
            address spender = approvals[i].spender;

            allowance[owner][token][spender].amount = 0;
            emit Lockdown(owner, token, spender);
        }
    }
}
```

In essence, Permit2 combines the power of meta transactions and EIP-712 structured data to provide a more gas-efficient and user-friendly token approval mechanism on EVM-based blockchain. It is especially useful in scenarios where users frequently interact with multiple tokens and spenders.

2. Flash Accounting in Uniswap V4

> The singleton design complements another architectural change in v4: flash accounting.
>
> In previous versions of the Uniswap Protocol, each operation (such as swapping or adding liquidity to a pool) ended by transferring tokens. In v4, each operation

> updates an internal net balance, known as a delta, only making external trans- fers at the end of the lock. The new take() and settle() functions can be used to borrow or deposit funds to the pool, respectively. By requiring that no tokens are owed to the pool manager or to the caller by the end of the call, the pool's solvency is enforced. Flash accounting simplifies complex pool operations, such as atomic swapping and adding. When combined with the singleton model, it also simplifies multi-hop trades. In the current execution environment, the flash accounting architecture is expensive because it requires storage updates at every balance change. Even though the contract guarantees that internal accounting data is never actually serialized to storage, users will still pay those same costs once the storage refund cap is exceeded. But, because balances must be 0 by the end of the transaction, accounting for these balances can be implemented with transient storage, as specified by EIP-1153.

In general, Flash Accounting is a method of accounting for token balances in a blockchain transaction. In traditional transactions, if there are multiple steps involving transfers of tokens, each transfer is executed and the token balances are updated after each step. This means that tokens are transferred into and out of the contract at each intermediate step, which costs gas, the fuel for computation on the Ethereum network.

In contrast, Flash Accounting only updates the net balances at the end of the transaction. The intermediate steps are calculated off-chain and only the final net change is recorded on-chain. This approach can reduce the gas cost significantly as it avoids multiple token transfers.

The advantages of flash accounting include:

- **Gas Savings** : Flash accounting can reduce the gas cost by avoiding multiple token transfers. This is particularly beneficial when interacting with multiple contracts in a single transaction.
- **Atomicity** : Since the final state is only recorded at the end of the transaction, it ensures the transaction is atomic. This means that all steps either succeed or fail together, which reduces the risk of partial completion.
- **Simpler Code** : With flash accounting, the code can be cleaner and easier to read and maintain as it avoids multiple token transfer calls.

However, from my point of view, Flash accounting is most efficient when used with transient storage, which is proposed in Ethereum Improvement Proposal (EIP) 1153. This

EIP is not yet part of the Ethereum protocol, and it's unclear when or if it will be adopted. (https://eips.ethereum.org/EIPS/eip-1153)

3. How do Hooks work?

> Uniswap v3 introduced us to concentrated liquidity in 2021. It was a very powerful, but also a highly opinionated, approach to handle liquidity on DEXes. With a lot of new features, gas fees was generally a bit higher compared to Uniswap v2, but allowed builders to create some amazing integrations.

> With Uniswap v4, they're offering more choice to the users. Hooks are essentially smart contracts that can "hook" into the lifecycle of a trading pool on Uniswap. Pools on v4 can then choose to make their own tradeoffs; for example, accepting a higher gas cost in exchange for a feature set similar to Uniswap v4, or add entirely new functionality.

> With this new customizability possible, it is no longer necessary to "fork" Uniswap to create a derivative DEX, a slightly different modified version of Uniswap that makes the tradeoffs you want it to make. Since you can just plug-in to different actions in a pool's lifecycle, you can just deploy a pool with your custom hooks and it will work alongside all the other Uniswap v4 pools as normal. At most, perhaps you want to build a custom frontend that guarantees your users will always be going through the pool with your hooks, instead of some other pool for the same token pair.

In Uniswap v4, hooks are implemented as contracts that can be associated with a pool at the time of its initialization. These hook contracts can implement any of the following callbacks in the lifecycle of pool actions:

- `beforeInitialize` and `afterInitialize`
- `beforeModifyPosition` and `afterModifyPosition`
- `beforeSwap` and `afterSwap`
- `beforeDonate` and `afterDonate`

How the hooks are created as follow:

```
function createHookMask({
  beforeInitialize,
  afterInitialize,
  beforeModifyPosition,
  afterModifyPosition,
  beforeSwap,
  afterSwap,
  beforeDonate,
  afterDonate,
}: HookMask): string {
  let result: BigNumber = BigNumber.from(0)
  if (beforeInitialize) result = result.add(BigNumber.from(1).shl(159))
  if (afterInitialize) result = result.add(BigNumber.from(1).shl(158))
  if (beforeModifyPosition) result = result.add(BigNumber.from(1).shl(157))
  if (afterModifyPosition) result = result.add(BigNumber.from(1).shl(156))
  if (beforeSwap) result = result.add(BigNumber.from(1).shl(155))
  if (afterSwap) result = result.add(BigNumber.from(1).shl(154))
  if (beforeDonate) result = result.add(BigNumber.from(1).shl(153))
  if (afterDonate) result = result.add(BigNumber.from(1).shl(152))
  return utils.hexZeroPad(result.toHexString(), 20)
}
```

These callbacks are functions that are executed at specific points in the pool's lifecycle. For example, the `beforeSwap` function would be executed before a swap operation takes place in the pool, and the `afterSwap` function would be executed after the swap operation.

Hooks can also specify fees on swaps or liquidity withdrawal. These fees are implemented using callback functions in the hook contract. The fee values or the logic of these callbacks can be updated by the hooks depending on their implementation. However, which callbacks are executed on a pool, including the type of fee or lack of fee, cannot change after the pool is initialized.

There are 4 examples explained hooks from main repo, in general, I can abstract like this:

```
contract MyHook {
    function beforeSwap(...) external {
        // Custom logic to be executed before a swap
    }    function afterSwap(...) external {
        // Custom logic to be executed after a swap
    }    // Additional functions for other hooks...
}
```

In this abstraction, `MyHook` is a hook contract that implements the `beforeSwap` and `afterSwap` hooks. The `...` in the function parameters would be replaced with the actual parameters required by these functions. The body of these functions would contain the custom logic that should be executed before and after a swap operation, respectively.

For more understand of hoos and practice, here is a good source:
https://learnweb3.io/lessons/uniswap-v4-hooks-create-a-fully-on-chain-take-profit-

orders-hook-on-uniswap-v4/

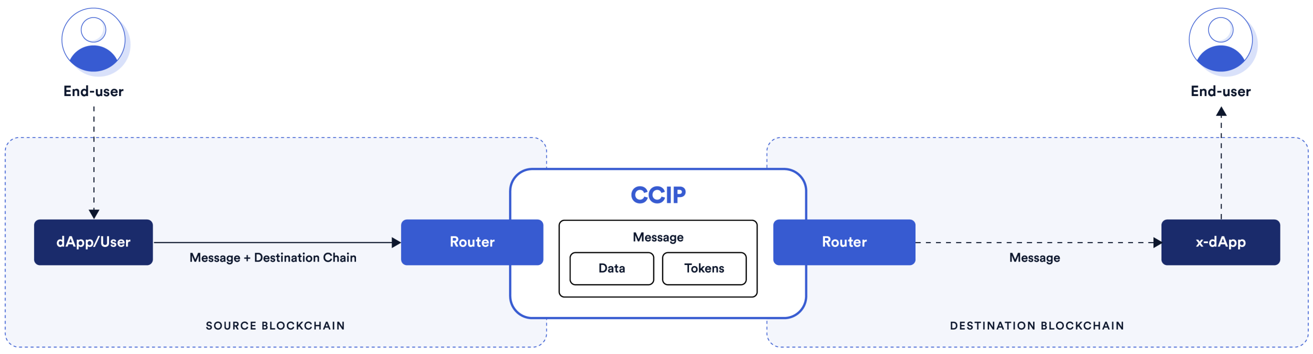
4. Crosschain and CCIP

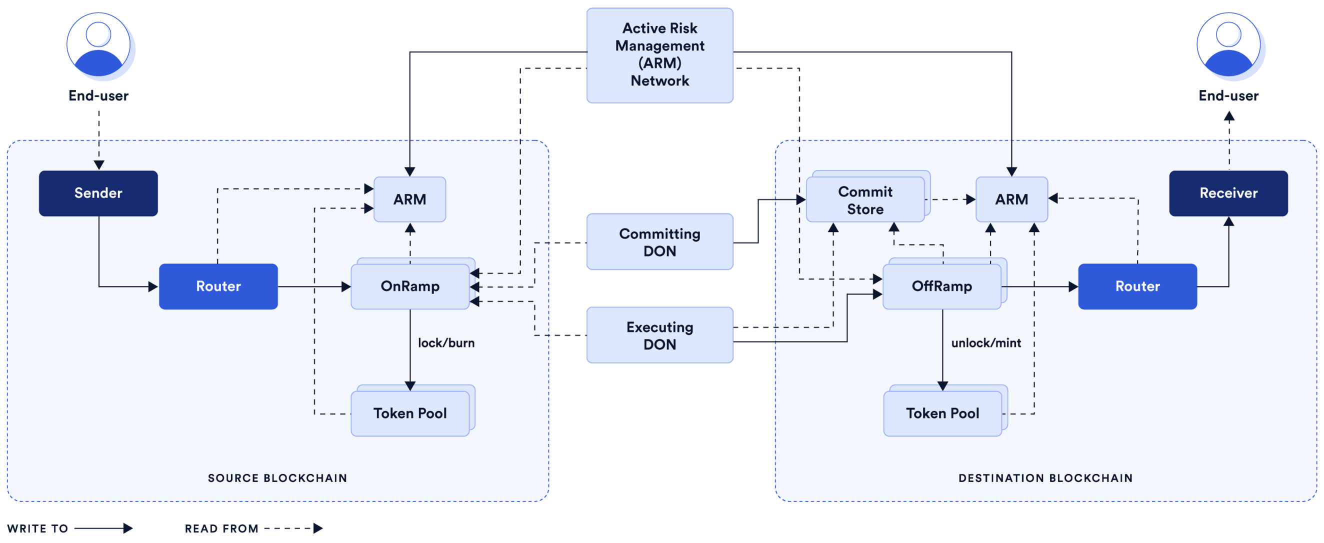
Cross-Chain

Cross-chain refers to the interaction between different blockchains. It enables the transfer of information or assets from one blockchain to another. This can be particularly useful because each blockchain may offer unique advantages, and being able to interact across chains allows users to leverage the best aspects of each one. Cross-chain can happen in several ways, including through the use of bridges, atomic swaps, or sidechains.

Chainlink CCIP

> The *Chainlink Cross-Chain Interoperability Protocol (CCIP)* provides a single simple interface through which dApps and web3 entrepreneurs can securely meet all their cross-chain needs, including token transfers and arbitrary messaging.





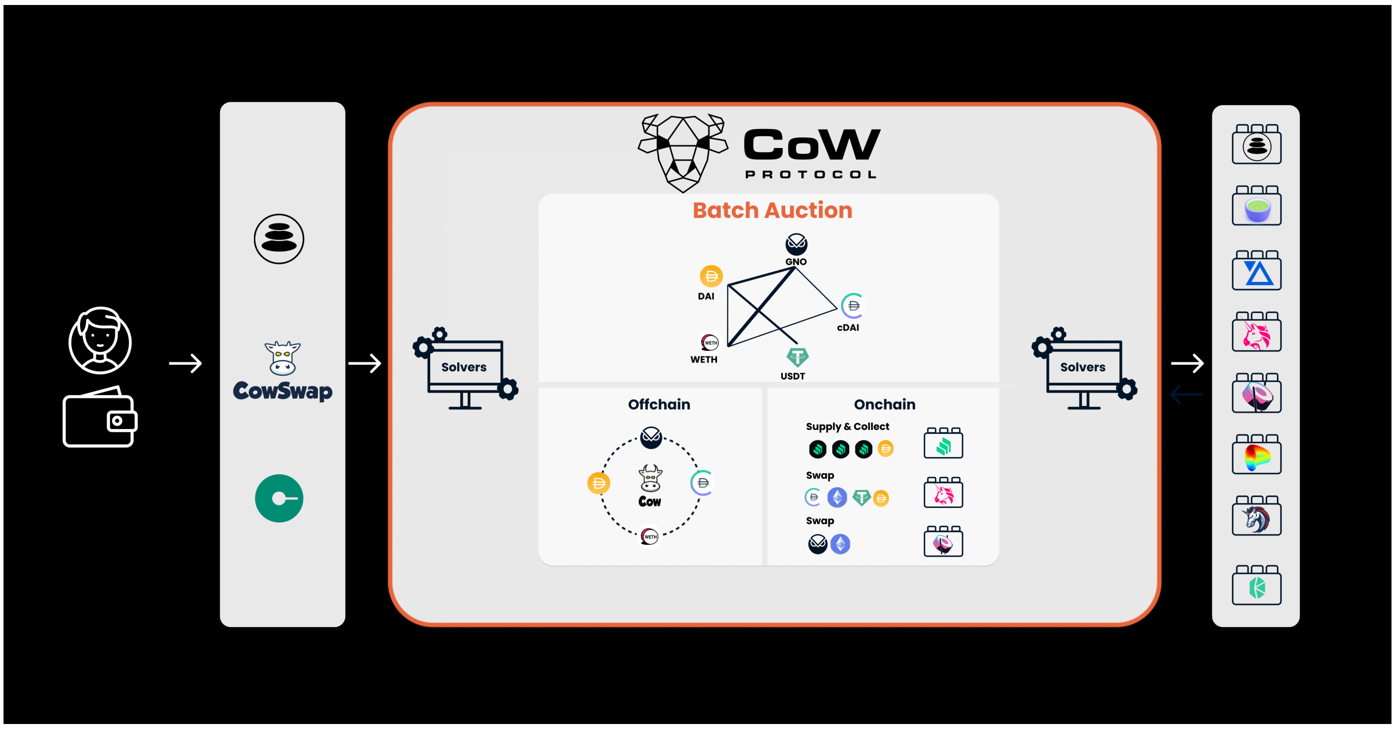
Okay: let's break down how Cross-Chain Interoperability Protocol (CCIP) works using an example as show in above figures. For simplicity, let's consider a cross-chain transaction of transferring tokens from Blockchain A (source) to Blockchain B (destination).

- **Step 1: Initiating the Transaction** : A user (sender) initiates a cross-chain transaction by interacting with the Router smart contract on Blockchain A. The transaction could be a function call, a token transfer, or sending arbitrary messages.
- **Step 2: Processing by OnRamp** : The Router forwards the transaction details to the OnRamp contract. The OnRamp contract verifies the transaction's validity based on Blockchain B's rules. It also checks the message size and gas limits, sequence numbers, and if the transaction includes tokens, interacts with the TokenPool.
- **Step 3: Committing DON Monitoring** : The transaction event is monitored by the Committing Data Oracle Network (DON). Once finality is achieved on Blockchain A, the Committing DON bundles the transactions and creates a Merkle root. The Merkle root is then signed by a quorum of oracle nodes part of the Committing DON.
- **Step 4: Storing in CommitStore** : The Merkle root is stored in the CommitStore contract on Blockchain B. However, the Merkle root needs to be 'blessed' by the Active Risk Management (ARM) network before the Executing DON can execute them on Blockchain B.
- **Step 5: ARM Network Blessing** : The ARM network, which consists of a set of independent nodes, monitors the Merkle roots committed in the CommitStore. After verifying that the committed Merkle roots match the transactions received by the OnRamp contract, it calls the ARM contract to "bless" the committed Merkle root.
- **Step 6: Executing DON** : Once the ARM network blesses the message, the Executing DON creates a valid Merkle proof, which is verified by the OffRamp contract against the Merkle root in the CommitStore contract. After these checks

pass, the Executing DON calls the OffRamp contract to complete the CCIP transactions on Blockchain B.

- **Step 7: Processing by OffRamp** : The OffRamp contract ensures the message is authentic by verifying the proof provided by the Executing DON. It ensures the transaction is executed only once. After validation, the OffRamp contract transmits any received message to the Router contract. If the transaction includes token transfers, the OffRamp contract calls the TokenPool to transfer the correct assets to the receiver.
- **Step 8: Completing the Transaction** : The Router on Blockchain B delivers the tokens or the message to the receiver's account or smart contract.

5. CoW Swap



CoW Swap is a decentralized trading platform built on top of the Coincidence of Wants (CoW) Protocol. It acts as a Meta DEX aggregator and aims to provide users with the best prices across various exchanges.

Okay, we can break it down into its core components and technologies:

- **Part 1: CoW Protocol:** This is the underlying technology for CoW Swap. It uses the concept of batch auctions and Coincidence of Wants (CoWs) to maximize liquidity. In simple terms, CoWs occur when two traders want what the other has, allowing them to swap their assets without going through liquidity pools. This process significantly reduces the cost for traders.

- **Part 2: Batch Auctions:** Unlike typical order book models or AMMs (Automated Market Makers), the CoW Protocol employs batch auctions. In this system, orders are collected over a certain period and then settled all at once. This mechanism helps in reducing the influence of arbitrageurs, ultimately offering a fairer price.
- **Part 3: Solvers:** Solvers play a crucial role in the CoW Protocol. They are responsible for proposing the most optimal settlement for the batch. This solution then gets chosen based on how much it can maximize traders' surplus. The winning solver gets rewarded with tokens.
- **Part 4: Off-chain Orders:** One significant feature of CoW Swap is its usage of off-chain orders. This means traders can place their buy/sell orders by just signing a message containing their trade details. This approach removes the necessity of paying gas fees for posting and cancelling orders.
- **Part 5: Gasless Trading:** CoW Swap is designed to minimize the need for gas payments. Orders are signed messages, and therefore do not require gas for submission. Only when the transaction is executed are gas fees incurred.

Beside, the concept of Batch Auctions is also

> On CoW Protocol, orders are placed off-chain and are not immediately executed, but rather collected and aggregated to be settled in batches. CoW Protocol replaces a central operator by an open solvers competition for order matching, with the term solver referring to anyone who submits an order settlement solution for a batch. As soon as a batch is "closed for orders" meaning that it does not consider any new orders, these solvers can compete to provide optimized solutions matching the orders in this closed batch.

But, why use batch auctions as a price-finding mechanism for a DEX? The two main reasons behind our development of batch auctions into a trading mechanism are:

- Give the DeFi space in Ethereum a chance to establish the same price of any token pair in the same block.
- Improve the DEX trading price offering by combining new economic mechanisms together such as Uniform Clearing Prices and Coincidence of Wants.

6. Uniswap and CowSwap

I am familiar with Uniswap because Uniswap is one of the most widely used and well-known DEX in crypto space. Uniswap has a larger userbased and well-established ecosystem. I believe if you are looking for a well-tested and widely adopted protocol with straighforward interfaces, Uniswap would be a good choice.

Beside, Uniswap, with its popularity, has significant liquidity for a large number of supported topkens, which can be lead to a better trade execution.

However, CoWSwap althought appears later, but they introduce a new mechanism for trading, with unique value propostion with its batch auctions and CoWs, and the potential for gasless transctions and miners with off-chain mechanism.

7. Blockchain development frameworks

I have been working in Blockchain Web3 systm design since 2017, so I would say I am familiar with quite a lot of development frameworks, e.g, hardhat, scaffold-eth, web3cli, Foundy development toolkit, truffle, brownie , and storages, ipfs, avwe, etc.

Part 2: Assignments

1. Understanding Uniswap v4

Question. Could you explain how Uniswap v4 functions? What innovative aspects does v4 bring to the table? Please also try to project the potential future innovations that could be introduced by v4.

Answer. Let see what's new in Uniswap v4,

- **Hooks** : At the heart of Uniswap v4 is a new concept known as 'hooks'. Think of hooks like plugins you can add to your music software to create a new sound effect. Similarly, these hooks can be used to add new functionalities or features to the liquidity pools in Uniswap v4. In practical terms, hooks can enable a variety of functions like setting up limit orders, dynamic fees or creating custom oracle implementations.
- **Singleton and Flash Accounting** : In previous versions, every new token pair had its own contract. However, Uniswap v4 introduces the singleton design pattern, meaning all pools are managed by a single contract. And Uniswap v4 uses a system called 'flash accounting'. This method only transfers tokens externally at the end of

a transaction, updating an internal balance throughout the process. All this improves gas costs a lot.

- **Native ETH** : Uniswap v4 is bringing back support for native ETH. So, instead of wrapping your ETH into an ERC-20 token for trading, you can trade directly using ETH. Another feature for saving gas.
- **ERC1155 Accounting** : With Uniswap v4, you can keep your tokens within the singleton (that mega contract we talked about earlier) and avoid constant transfers to and from the contract. The accounting itself uses the ERC1155 standard which is a multi-token standard. It allows you to send multiple different token classes in one transaction.
- **Governance Updates** : Uniswap v4 also brings changes to how fees are managed. There are two separate governance fee mechanisms - swap fees and withdrawal fees. The governing body can take a certain percentage of these fees.
- **Donate Function** : Uniswap v4 introduces a `donate` function that allows users to pay liquidity providers directly in the tokens of the pool.

Not a future prediction, recently Uniswap Labs has released a new protocol called UniswapX.

> X the protocol is a collection of open source, immutable smart contracts. The protocol's core contracts are known as Reactors. At a high level, Reactors interpret trade orders and coordinate the distribution of tokens between counterparties according to rules encoded in those orders. While they are highly configurable contracts, each Reactor contract can only interpret one order type.

> Reactors make use of signed orders, which necessitates a change in the transaction flow for Swappers. Instead of constructing a trade transaction and broadcasting it to the mempool themselves, Swappers instead just add their signature to a piece of offchain data that contains the terms of a trade that they would be willing to do; someone else executes it for them.

> Reactor contracts can manage the distribution of fees on a trade-by-trade basis. Fees can be defined in two ways:

- > The owner of the Reactor (currently the Uniswap Timelock) can configure a `ProtocolFeeController` contract. That contract's configuration determines the criteria of trades that should be charged a fee, the address to which the fees are sent, and the rate (no greater than 5 basis points of the

> output token). A governance vote is required to enable the `ProtocolFeeController` contract.

- > Orders can be configured (most likely by user interfaces) to send fees of any amount of the output token to any address. Each order can have an arbitrary fee configuration (orders can have any number of outputs), and the swapper must sign the order for it to be valid.

Source: https://gov.uniswap.org/t/uniswapx-and-the-dao/21718

2. Securing Cross-Chain Transactions

Q. How can we ensure the security of cross-chain transactions? Among Layer0 and Chainlink (CCIP), which one do you think provides a better solution for cross-chain interoperability and why? Please elaborate on your answer, considering various factors such as security, scalability, and performance.

A. Securing cross-chain transaction could be a challenge task, because there is no central entity to gurantee transaction integrity between chains. There are some common methods to guarantee the security of cross-chain trasnactions, e.g, hash-locking, time-locking, incentive/penaties.

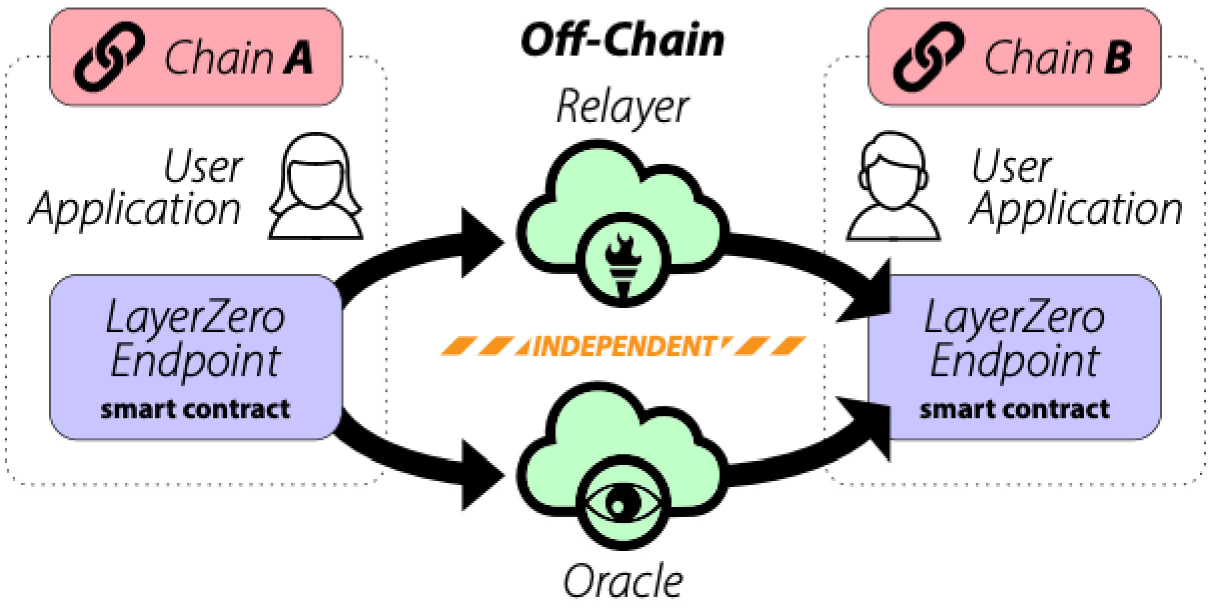
Among Layer0 and Chainlink CCIP, we can go with the definition first:

Layer0:

> LayerZero, the first trustless omnichain interoperability protocol, which provides a powerful, low level communication primitive upon which a diverse set of cross-chain applications can be built. Using this new primitive, developers can implement seamless inter-chain applications like a cross-chain DEX or multi-chain yield aggregator without having to rely on a trusted custodian or intermediate transactions.

The first and easiest use case for LayerZero to go after is bridging. Bridges are a nightmare, and involve reasonable risk, LayerZero bridging infrastructure (being used by Stargate) aims to reduce this. Bridging is not a nice experience - going from one EVM chain (say Ethereum) to another (say Fantom) can be a struggle. Current bridges suffer from stuck transactions, lack of liquidity, slippage and high fees, not to mention smart contract risk or hacks. Alternatives include going via CEXs on occasion, but that's not always possible and is obviously less secure ('not your keys, not your coins').

How does it works?



> LayerZero Endpoint exists on each (supported) chain, and any chain with a LayerZero Endpoint can conduct cross-chain transactions involving any other chain with a LayerZero Endpoint. In essence, this creates a fully connected network where every node has a direct connection to every other node. With minor boilerplate code, **any blockchain is supported.**

We have discussed about Chainlink CCIP above.

As for the comparison between LayerZero and Chainlink's (CCIP), I believe that both solutions bring their unique approaches and strengths.

Chainlink's CCIP, for instance, is built on the strong foundation of Chainlink's widely adopted decentralized oracle network, which could instill greater confidence in the system's security and reliability. Chainlink also has strong relationships with numerous blockchains and DApps, which could aid in adoption. However, CCIP may be more complex to implement than Layer0 due to the intricate oracle network.

On the other hand, Layer0 offers a light and efficient protocol that can theoretically work with any blockchain with minor boilerplate code. The protocol's flexibility could make it easier for DApps to adapt LayerZero to their specific needs. However, it may face adoption hurdles due to its novelty and reliance on external entities like Oracles and Relayers.

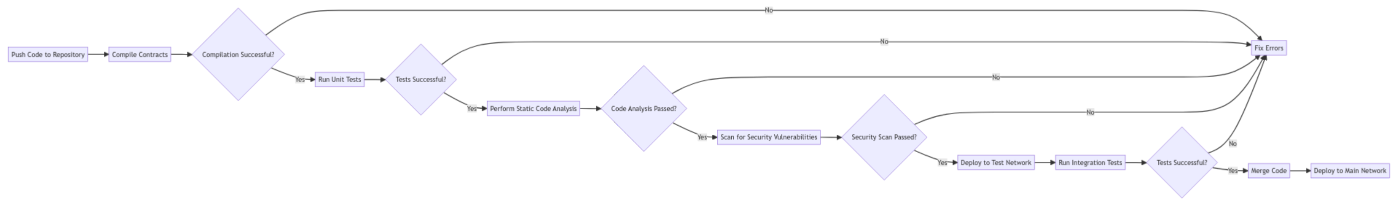
In terms of scalability, both Layer0 and Chainlink (CCIP) aim to increase the scalability of blockchain networks, but in different ways. Layer0 seeks to create a fundamental layer that can support any blockchain, potentially allowing them to interact more seamlessly. Chainlink, meanwhile, focuses on off-chain computation and data provision to alleviate the load on-chain.

In terms of performance, it's hard to compare the two directly since they offer different functionalities and are at different stages of development. In my opinion, Chainlink's oracle networks have been performing well in the real world, while Layer0's performance remains to be seen as it's still in the early stages of development.

3. Continuous Integration for Smart Contract Development

Q. Design a process for the automatic continuous integration (CI) of smart contract development. Here's a twist: how could we ensure the same address is retained when delivering new features, without considering upgradable contracts? Elaborate on any tools, technologies, or strategies you would use in this process.

A. From my experiece, designing a Continuous Integration (CI) process for smart contract development should automate the following steps: compilation, testing, static code analysis, security vulnerability scanning, and deployment. The simple workflow could be looked like below:



Some needed tools:

- Source Control System: Git
- CI/CD Tool: Jenkins, CircleCI, Travis CI, etc.
- Development Environment: Truffle Suite (Includes Ganache, Truffle)
- Static Code Analysis Tool: Solhint or Mythril for Solidity
- Security Vulnerability Scanning Tool: Slither, MythX, Securify
- Testing Framework: Mocha and Chai

System workflow:

- **Step 1: Push Code to Repository** : Developers write smart contract code and push it to a version control system like Git. This should trigger the CI process.
- **Step 2: Compile Contracts** : The CI tool pulls the latest code from the repository and compiles it. This step helps to identify any syntax errors or bugs that can be caught at compile time. Tools like Truffle can be used for this purpose.
- **Step 3: Run Unit Tests** : Once the smart contract is successfully compiled, the CI tool runs unit tests against the compiled contract. You can write tests using a framework like Mocha and Chai. Running tests after every commit helps catch issues early in the development process.
- **Step 4: Perform Static Code Analysis** : The CI tool performs static code analysis using tools like Solhint or Mythril. This helps enforce code quality and catch potential bugs that were not caught during the testing phase.
- **Step 5: Scan for Security Vulnerabilities** : The CI tool scans the smart contracts for known security vulnerabilities using tools like Slither, MythX, or Securify. This step is crucial due to the immutable nature of smart contracts.
- **Step 6: Deploy to Test Network** : If the smart contract passes all tests, code analysis, and vulnerability scanning, it is deployed to a test network using tools like Truffle.
- **Step 7: Integration Testing** : Now on the test network, integration tests are run. These tests should involve interaction between multiple smart contracts or even multiple blockchain systems.
- **Strep 8: Report** : Any errors or warnings are reported back to the developers. If any tests fail, the build should fail and the developers should be alerted.
- **Step 9: Merge** : If all tests pass, the changes are ready to be merged into the main branch.
- **Step 10: Deploy to Main Network** : After successful testing and merging of the code into the main branch, the contracts are deployed to the mainnet.

  Regarding the question howto retain the same address when delivery new features for smart contracts, from my knowledge, we can use proxy patterns, whcih can be used to call the logic from another contractwhile maintining their own state, so you can change the logi c of the contract without changing the address of the proxy contracts. In addition, instead of adding features directly into existing contract, you could develop new contracts that interact with the existing contract. The new contracts hold the new feautres and logisc, but not affact the address of original contract. Finally, we can use external libraries as a functions, solidity allows contract to call these functions.

4. Continuous Integration for Smart Contract Interface (ABI) with Other Projects

Q. Please outline a process for the automatic continuous integration of a smart contract interface (ABI) with other projects. Consider an example where a front-end repo uses the ABI to call the smart contract, and this ABI is subject to frequent changes due to the development phase. How would you ensure a smooth and efficient process?

A. Similar to the above answer, I would like to emphasize the process of design and workflow as below:

Tools/frameworks required:

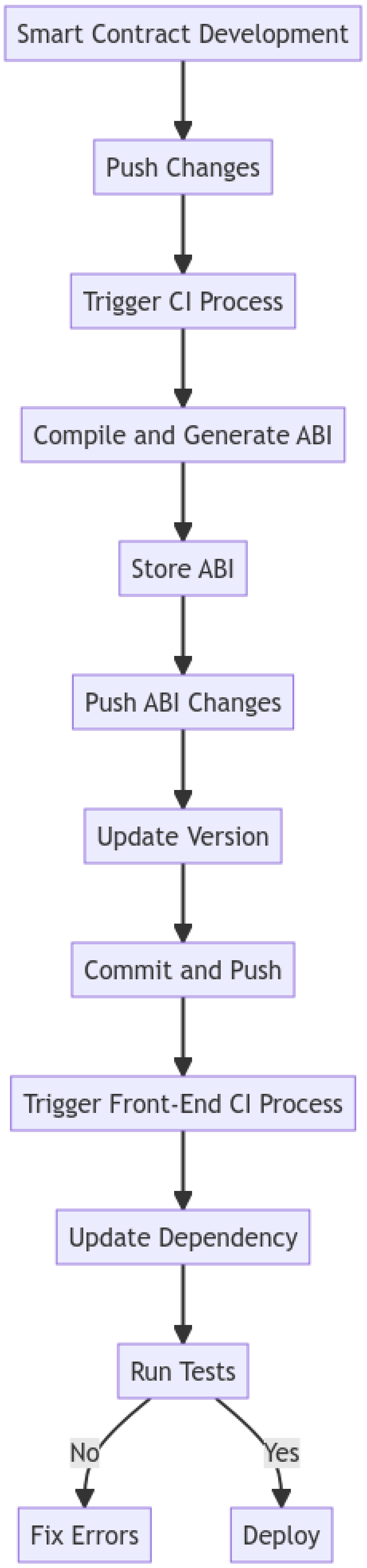
- Source Control System: Git
- CI/CD Tools: Jenkins, CircleCI, Travis CI, github Action (I am familar with jenkins, Circile CI, Github Action)
- Package Manager: npm or yarn (for front-end applications as required)
- Smart Contract Development Framework: Truffle Suite (for this example)

Workflow:

- **Step 1: Smart Contract Development** : Developers make changes to the smart contract, which may result in changes to the ABI.
- **Step 2: Push Changes** : Developers push these changes to the repository.
- **Step 3: Trigger CI Process** : The pushing of changes triggers the Continuous Integration (CI) process.
- **Step 4: Compile and Generate ABI** : The CI server compiles the updated smart contract and generates the new ABI.
- **Step 5: Store ABI** : The new ABI is stored in a dedicated directory within the project or in a separate repository specifically created for the ABIs. It can be stored

in a JSON file format which is typically used for ABIs.

- **Step 6: Push ABI Changes** : The changes to the ABI are pushed to the repository.
- **Step 7: Update Version** : The version of the ABI in the package.json (or similar file) is updated, following semantic versioning principles.
- **Step 8: Commit and Push** : The updated package.json file is committed and pushed to the repository.
- **Step 9: Trigger Front-End CI Process** : The front-end repository is set up to trigger its own CI process whenever there is an update to the ABI repository.
- **Step 10: Update Dependency** : The front-end application's CI process updates the ABI dependency to the new version using npm or yarn.
- **Stpe 11: Run Tests** : The CI server runs tests on the front-end application to ensure the new ABI hasn't broken anything.
- **Step 12: Deploy** : If all tests pass, the new version of the front-end application is deployed.


Smart Contract Development
Push Changes
Trigger CI Process
Compile and Generate ABI
Store ABI
Push ABI Changes
Update Version
Commit and Push
Trigger Front-End CI Process
Update Dependency
Run Tests
No
Yes
Fix Errors
Deploy

References

[1] https://github.com/Uniswap/permit2/tree/main

[2] https://blog.uniswap.org/permit2-and-universal-router?utm_source=tldrnewsletter

[3] https://github.com/Uniswap/v4-core/tree/main

[4] https://docs.cow.fi/

[5] https://docs.chain.link/ccip