André Zommerfelds
August, 2024

Development and launch plan for an interoperability feature in a crypto platform

Management summary

In the current crypto market, interoperability between blockchain networks is essential for bridging or swapping assets and enabling arbitrary crosschain communication. This project aims to develop a crosschain messaging protocol that facilitates seamless communication and asset transfer between blockchains.

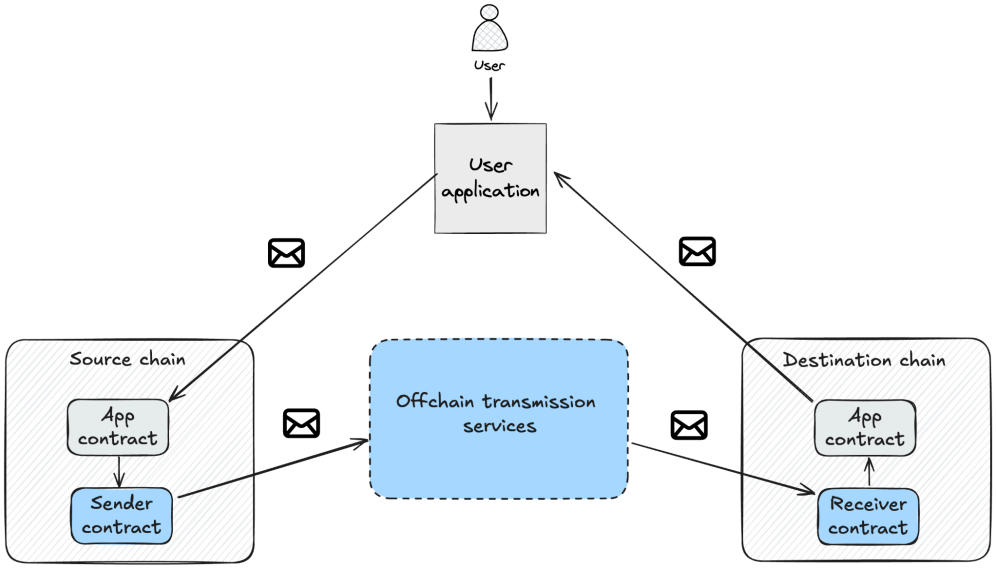
The document outlines the development plan for an interoperability protocol, which includes key components like sender contracts, attesters, relayers, block header oracles, and receiver contracts. The initial focus is on EVM-based blockchains due to their significant liquidity and fragmentation. The development process is estimated to require roughly 30 person weeks, which can be distributed and parallelized with a small team of engineers and testers, and a product manager to ensure target customer discovery is continuously integrated.

The approach includes starting with a permissioned network of attesters and relayers, with plans to gradually decentralize the network post-mainnet launch and make it fully open-source and permissionless. This strategy balances security and scalability, inspired by best practices from existing crosschain bridges and leveraging zero-knowledge proofs for minimizing trust.

1. Feature specification

1.1. General implementation of crosschain messaging protocols

Crosschain messaging protocols generally implement the following flow, starting with a message containing information for app contracts to use (e.g. swap ETH on Ethereum to USDC on Solana):



Source: own illustration

Message flow:

1. **User**: submits message with instructions to source chain smart contract
2. **Source chain & sender contract**: accepts & stores message in a block
3. **Transmission services**: sends package with necessary proofs to destination chain
4. **Destination chain & receiver contract**: receives package, verifies proofs, and forwards instructions to any other smart contract containing further business logic

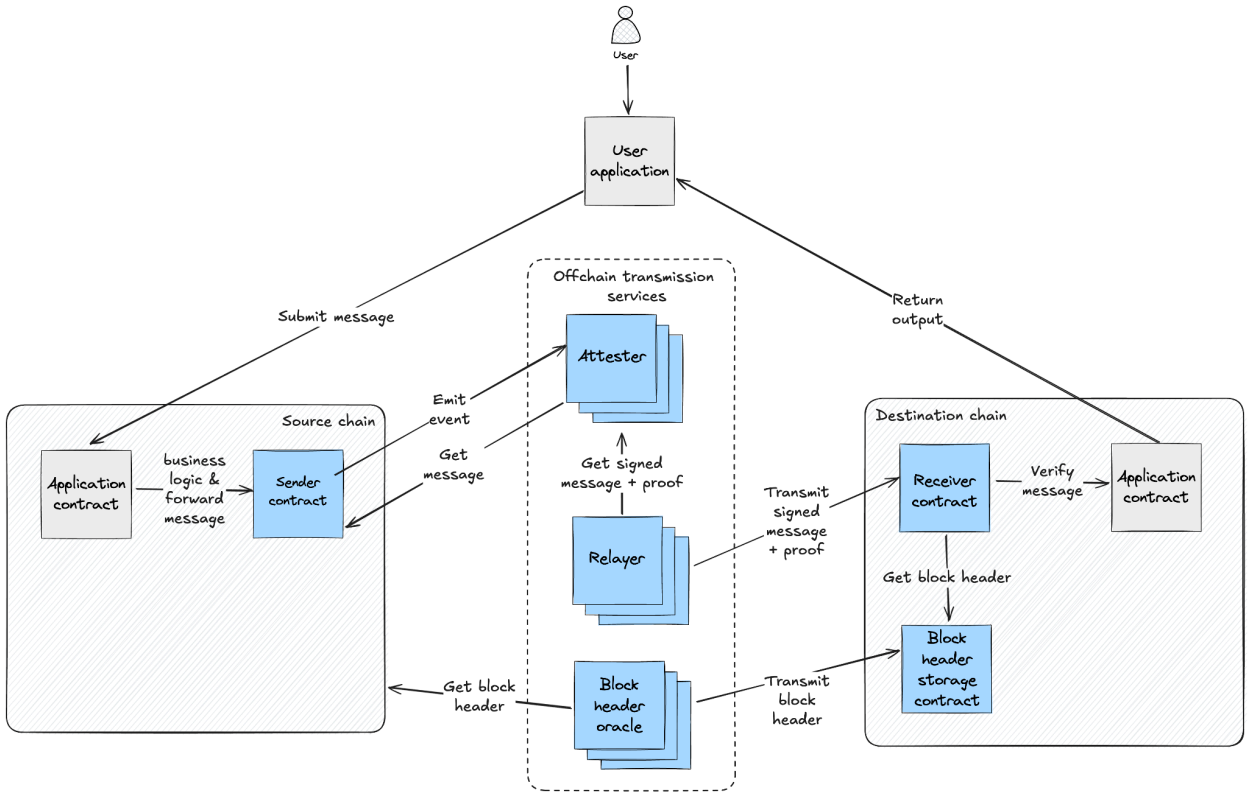
Crosschain interoperability protocols implement different components in different forms and trust assumptions. These variations can be categorized with the way they transmit messages, prove state inclusion in the source chain to the destination chain, and ensure censorship resistance. One way to categorize them is using the Crosschain Risk Framework and the work done by L2beat. See appendix 8.1. Current crosschain messaging ecosystem for an overview of existing crosschain messaging solutions and how they are categorized using that framework.

1.2. Proposed architecture (high-level)

Like most protocols mentioned above, the proposal is to implement a third-party attestation protocol. There are two main tasks that such an interoperability protocol must support:

- Checking the validity of the information transmitted
- Transmitting the information from one chain to the other

These tasks can be distributed to different components or actors in the network, as described in the general implementations. The following high-level architecture highlights in **blue** which components will make part of our proposed interoperability protocol to achieve the tasks above.



Source: own illustration

Message flow

1. **User** submits a message
2. **Application contract (source chain)** executes any business logic and passes the message to the transmission contract
3. **Sender contract** emits a corresponding event
4. **Attester** listens event, takes message and corresponding block header to create merkle proof (tx hash, sibling hashes, merkle root) and sign the package
5. **Relayer** takes signed attestation and forwards it

6. **Receiver contract** validates the received package by verifying merkle proof and comparing block headers using the **block header storage contract** (which gets data from **block header oracles**)
7. **Application contract (destination chain)** takes validated message and executes further business logic

Notes

- Attester should run a full node from the source chain to be able to check the validity of the message transaction and create the corresponding merkle proof
- Block header oracle can be a network that provides block headers from the source chain to the destination chain (similar to LayerZero), but we propose the usage of an oracle that produces a succinct proof (e.g. zk-SNARKs) such as a zkOracle
- Block header storage contract makes the source chain block header available once a certain consensus threshold is reached

1.3. Key functionalities & user stories

See the full overview of requirements in the appendix 8.2. Requirements.

Component	Function	User story
Sender contract	Integrates with smart contracts to emit and store messages on source chain	As a developer, I want to integrate my smart contract with a contract to securely handle crosschain messages
Attester	Generates and signs merkle proof to verify message validity	As a developer, I want transmitted message to be trusted and verifiable
Relayer	Delivers message package and facilitates modular crosschain messaging	As a developer, I want protocol to be modular, with different nodes focusing on transmission and attestation
Receiver contract	Verifies merkle proofs and block headers on destination chain	As a developer, I want message on destination chain to be verified for authenticity
Block header oracle	Provides secure and timely block headers from source chain	As a developer, I want secure and accurate block headers for message validation
Block header storage contract	Stores and provides trusted block headers for validation	As a developer, I want secure and accurate block headers for message validation

1.4. Reason for the chosen implementation

The above implementation aims to take the best characteristics of existing solutions:

- **Separation of attesters and relayers**: similar to the Wormhole architecture, we implement the separation of attesters and relayers. This aims to make the protocol modular, which enables the separation of duties and allows for high specialization.
- **Separate block header oracle**: similar to LayerZero and Chainlink's CCIP, the usage of an oracle to provide block header, independently of the attester/relayer, should lead to an increase in security of the protocol and allows for specialization as argued above.
- **Zero-knowledge proofs from block header oracles**: similar to zkOracle or zkBridge, we propose the usage of zero-knowledge proofs to reduce trust for a smart contract in the destination to check the validity of the block headers received from the oracles.

1.5. Target blockchains for initial implementation

In an initial stage, the above setup will be implemented for EVM-based blockchains, because of their existing liquidity and the need to solve respective liquidity fragmentation between L2s.

At a later stage, we plan to phase out to other VMs, by prioritizing over existing demand (developers, users, liquidity) vs. complexity to build.

2. Development plan

2.1. Project size and timeline

We identify the following phases for the development project, along with rough effort estimations in person weeks (pw) for each subtask:

Phase	Subtasks	Person weeks (est.)
Product design	Technical architecture	2
	Economics / Incentives design	2
	Requirements specification	2
Development	Sender contract	2
	Receiver contract	2
	Attester node	3
	Relayer node	2
	Block header oracle	3
	Block header storage contract	3
Audit	Security audit	3
Testing	Testing (unit, integration, security)	4
Rollout	Rollout buffer (local-testnet-mainnet, monitoring, bug fix, etc.)	4
Total person weeks		32
Total est. duration with 4 devs, 1 tester, 1 PM (person weeks)		**21**

See the full overview of requirements in the appendix 8.2. Requirements.

2.2. Roles and responsibilities

The initial development team would consist of:

- Product manager
- Tech lead / software architect
- 3-4 software and/or blockchain engineers, incl. expertise in cryptography and ZKPs
- Test engineer (QA)
- Security audit (external)

2.3. Risks and challenges

Here are the biggest risks and challenges identified:

Risk or challenge	Description
Smart contract vulnerability	Bugs or vulnerabilities in smart contracts can lead to significant financial losses
Technological complexity	High complexity in interplay between smart contracts, nodes, oracles and technologies like ZKP
Oracle manipulation	A compromised oracle network could lead to incorrect block headers being used (ZKP should prevent that)
Lack of economic incentives	Decentralization only happens if multiple parties want to participate in network of attesters, relayers, oracles
Prohibitively high onchain costs (gas fees)	Posting message or block headers with proofs in destination chain may come with high costs
Market trends	Current trend towards "intent-centric" bridges (catering solver layer) could make a "messaging-centric" solution less relevant
Competition	Existing or upcoming solutions could significantly adoption of our technology

3. Sprint planning

3.1. The first two sprints

Assuming a two-week sprint and that the product design phase is over, we would start with building the smart contracts, as they are the core components enabling the sending and receiving of messages from and to chains.

Sprint 1 (2 weeks)				
Component	**User story**	**Requirement**	**Identifier**	**PBI**
Sender contract	As a developer, I want to be able to connect smart contracts from different chains by integrating with a crosschain messaging protocol.	Functional	sender-contract-1	Contract interface: accept messages from application contract after executing necessary business logic.
Sender contract		Functional	sender-contract-2	Emit an event with message details, transaction hash, and metadata for attester.
Sender contract		Functional	sender-contract-3	Store message and associated data until successfully processed by attester and relayer.
Sender contract		Non-functional	sender-contract-4	Implement security features such as message nonce (replay attack)
Sender contract		Non-functional	sender-contract-5	Optimize event emission and storage for gas efficiency.

Sprint 2 (2 weeks)				
Component	**User story**	**Requirement**	**Identifier**	**PBI**
Receiver contract	As a developer, I want message on destination chain to be verified / trustworthy.	Functional	receiver-contract-1	Verify merkle proof provided by relayer by recalculating merkle root.
Receiver contract		Functional	receiver-contract-2	Validate block header by checking consistency against data stored in block header storage contract.
Receiver contract		Functional	receiver-contract-3	Verify attestation's signature to ensure it was created by a trusted attester.
Receiver contract		Functional	receiver-contract-4	Ensure that message, merkle proof, and block header correspond correctly.
Receiver contract		Non-functional	receiver-contract-5	Implement security measures to prevent malicious or fraudulent submissions.
Receiver contract		Non-functional	receiver-contract-6	Optimize contract operations for gas efficiency, particularly in proof verification and signature checking.

3.2. Criteria for moving tasks from the backlog to the sprint

Adding tasks to the sprint requires backlog prioritization, for which these factors can be used:

Criteria	Description
Criticality of features	Backlog items can be ordered by finishing core components first, including sender and receiver contracts as well attester and relayer. Block header oracles and storage contract, are crucial for decentralization, but not necessary for core functionality.
Readiness of design and specifications	It may be that functionalities of sender and receiver contracts are much simpler and clearer than how attester is built. In that case, we can start building contracts first.
Path dependencies	Smart contracts are predecessors for attesters and relayers to build upon.
Risks and challenges	Since smart contracts are most security critical, it is a good idea to start with them first. Especially if security audits also take some time.
Stakeholder requirements	It might be a requirement that core contracts are built first, and rest can be implemented even by different parties. We will just provide one way to do attestation and relaying between contracts.

3.3. Metrics and KPIs to measure sprint success

In a very early stage project, it might be harder to find “outcome” (instead of output) metrics that are relevant to track. Output metrics could be:

- Velocity (total items completed)
- Burndown chart
- Team satisfaction

More output focused metrics could be:

- Time to deliver a message
- Gas cost to transmit a message
- Number of successful messages transmitted
- Early user integrations
- Early user satisfaction

4. Integration and testing

4.1. Testing strategy

The testing strategy consists of cycling through the **testing plan (high-level)** below for different environments and testing dimensions (see below). Testing should be iterative based on test results and insights from the security audit, and the iteration should be part of an automated pipeline (continuous integration), especially for unit and integration testing.

Environments:

1. Local blockchains (e.g. using Hardhat Network)
2. Testnet (e.g. using Sepolia on Ethereum and Arbitrum)
3. Mainnet

Testing dimensions:

- Functionality testing
- Security testing (incl. fuzz testing)
- Onchain costs
- Performance testing
- Load & stress testing
- Regression testing (in case of changes)

4.2. Testing plan (high-level)

The following high-level testing plan is followed with the above-mentioned environments and testing dimensions:

1. **Unit testing:** validate individual functions and components
2. **Security testing:** detect vulnerabilities, stress testing, and code audits
3. **Integration testing:** ensure different components interact correctly across components
4. **E2E testing:** validate the entire business workflow from start to finish
5. **User acceptance testing (UAT):** check with a real business application whether the protocol meets business needs and user/developer expectations, especially regarding performance, cost, and security.

5. Documentation and tutorials

5.1. Documentation structure

Below is a proposed structure for our documentation:

Section	Content
Introduction	Overview: introduction to the protocol, key features, and use cases
Getting started	Quick start guide: setup, installation, contracts, first crosschain message
	Supported networks: mainnet and testnet deployment, network & contract configuration
Core concepts	Architecture overview: protocol components and message flow
	Security model: attester, relayer, and block header validation
Guides	Basic integration: implementing crosschain messaging, integrating with dapps
	Advanced features: running your own attester or relayer
Deployment	Deploying to testnet and mainnet: step-by-step deployment instructions
	Contract upgrades: managing migrations and updates
API reference	Smart contract ABI: function definitions, usage examples, event reference
	Relayer and attester interfaces: communication standards, example implementations
Resources	Tutorials and examples: crosschain token transfers, sample projects, starter kits
	Glossary: definitions of key terms
	Support: FAQ, troubleshooting, community support channels

5.2. Tutorial structure

Here is an overview of a tutorial structure with the example of a “burn & mint” smart contract:

Section	Content
Introduction	Purpose and prerequisites: guide through implementing crosschain token transfer
Setup	Tools installation: install necessary tools (e.g., Hardhat) Clone repository: clone tutorial repository from GitHub Deploy contracts: deploy protocol’s core contracts on local testnet using Hardhat
Implementing token contracts	Burn contract: create contract with burn functionality on source chain Mint contract: create contract with mint functionality on destination chain Event listeners: set up event listeners for token burn events on source chain and mint events on destination chain
Crosschain messaging	Sender contract integration: integrate burn contract with sender contract in source chain Attester: deploy local attester to generate and sign merkle proofs for burn event Relayer: deploy local relayer to transmit messages to destination chain Receiver contract: integrate mint contract with receiver contract on destination chain
Deploying to test or mainnet	Test/mainnet deployment: deploy burn contract to Ethereum/Arbitrum test/mainnet Test/mainnet execution: execute crosschain token transfer between chains

6. Launch strategy

6.1. Marketing strategy and milestones

Here is an overview of the launch strategy, marketing plans, milestones and steps for success:

Phase	Marketing plans	Key milestones	Steps for success
Testnet launch; First 3-4 months	- Pre-launch announcement on crypto social (X, Farcaster, TG, Reddit, Discord) and developer platforms (e.g. GitHub) - Publish technical blog posts on Medium, Mirror, Paragraph - Engage with crypto communities on crypto social, do AMAs - Organize webinars/podcasts in crypto social	- Core development completed - Security audits passed - Testnet launched - Onboarded early devs/builders	- Finalize core features and test on testnets - Deploy contracts and invite early developers - Initially, attester and relayer will be centralized and permissioned for security reasons - Monitor performance, gather feedback, and resolve issues
Mainnet launch; Next 3-4 months	Same as above, additionally: - Launch partnerships with established crypto projects - Participate in major crypto events (e.g., ETHGlobal, Consensus, Devcon) - Host developer workshops and hackathons to promote protocol adoption	- Implemented feedback from testnet launch - Mainnet launched - First apps built using protocol	- Address testnet feedback and finalize product - Scale infrastructure, ensure robustness, and prepare for decentralization - Launch marketing campaign and open access to all developers
Decentralization; Next 3-4 months	Decentralization roadmap: announcements on social media, blog posts, and community engagement	- Attesters and relayers are permissionless - Fully open-source code	- Open-source all code - Progressively opening attester and relayer network - If above OK, make them permissionless

6.2. Risks and mitigation strategies

Here is an overview of the risks and mitigation strategies associated with each phase:

Phase	Risks	Mitigation strategies
Testnet launch; First 3-4 months	- Vulnerabilities or security flaws - Technical complexity - Economic return (business viability)	- Extensive testing and audit; bug bounties - Engage communities early, create compelling incentives
Mainnet launch; Next 3-4 months	- Vulnerabilities or security flaws - Low user adoption - Technical failure due to unexpected behaviors in mainnet	- Extensive testing and audit; bug bounties - Engage communities early, create compelling incentives
Decentralization; Next 3-4 months	- Technical feasibility for pemissionlessness - Lack of economic incentives	- Scale and open-source infrastructure gradually

7. Customer and developer feedback

7.1. Strategy to gather and incorporate feedback

The following three points summarize how we define the target client, find opportunities and integrate them into the development.

1. **Define target customers:**
    - **Blockchain developers** building interoperable applications across multiple blockchains
    - **Sophisticated crypto builders** interested in being attesters, relayers, or contributing to the decentralized network
2. **Define & prioritize opportunities:**
    - Understand main needs, problems, or desires from target customers (e.g. when dealing with interoperability, or operating attesters/relayers)
    - Save them with user story / experience mapping
3. **Continuous client interaction:**
    - Weekly interviews: conduct short, focused interviews with target users every week
    - Build in the open: engage with the community through GitHub issues, comments, and X feedback
4. **Incorporate feedback:**
    - See “Analyzing and integrating feedback”

7.2. Channels and methods for collecting feedback

Here is a list of channels and methods for collecting feedback from our target clients:

- Discord or TG for technical Q&A
- GitHub for open discussions
- X for community-driven insights
- Webinars/AMAs/podcasts for attention
- Hackathons to encourage innovation and active feedback through events
- (Bug) bounties for security feedback

7.3. Analyzing and integrating feedback

1. **Prioritization:**
    - We prioritize the opportunities identified (e.g. from interviews & UX mapping) by:
        i. Size: how many target customers have reported it
        ii. Satisfaction: how important it is to their satisfaction
        iii. Business: how well it aligns with our business strategy
        iv. Market: how we want to position ourselves in the market
2. **Continuous discovery, continuous delivery:**
    - Feedback reviews: regular team syncs to discuss weekly feedback
    - Rapid iterations: deploy small updates regularly, based on direct user feedback
    - Adjust roadmap: incorporate continuous feedback into the long-term plan
    - Communicate changes & follow-up: update the community on how their feedback was implemented via channels mentioned above

8. Appendix

8.1. Current crosschain messaging ecosystem

Crosschain interoperability protocols implement different components in different forms and trust assumptions. These variations can be categorized with the way they transmit messages, prove state inclusion in the source chain to the destination chain, and ensure censorship resistance. One way to categorize them is using the Crosschain Risk Framework and the work done by L2beat:

Protocol type	Details
State validating protocols (trustless)	- Validation of state with full nodes - Hashed timelock contract (HTLC) for p2p asset exchange
Consensus verifying protocols (spectrum: trust-minimized -> trusted)	- Validation of consensus with light-client in smart contract - Validation of consensus with light-client in chain of chains (e.g., IBC) - Validation with a ZKP + light clients using succinct properties (e.g., SNARKs)
Third-party attestation protocols (spectrum: trusted -> trust-minimized)	- Multisig or threshold signature attestation - TEE, SGX attestation - Oracles / oracle networks attestation - Proof of authority (PoA) or proof of stake (PoS) attestation - Central authority attestation
Optimistic protocols (spectrum: trusted -> trust-minimized)	- (Permissionless) Attestors + watchers for (permissionless) fault-proof (1 of N trust assumption)
Hybrid protocols	- Combining character

Source: overview inspired by the Crosschain Risk Framework

Below are some examples of popular crosschain messaging protocols and how they may be categorized according to the framework used above. Our proposed solution leverages the best technologies used by these existing crosschain protocols.

- LayerZero (third-party attestation protocol)
- Wormhole (third-party attestation protocol)
- Chainlink CCIP (third-party attestation protocol)
- Cosmos IBC (consensus verifying protocol; onchain)
- zkBridge / Polyhedra Network (consensus verifying protocols; validity proof)
- Hashi (hybrid protocol; bridge aggregator)
- Across (hybrid protocol; settlement layer for asset transfer intents)

8.2. Requirements

Component	User story	Requirement Type	Identifier	PBI
Sender contract	As developer, I want to be able to integrate my smart contract containing business logic to crosschain smart contract, to manage messages from application and ensure they are securely emitted, and stored for crosschain communication.	Functional	sender-contract-1	Contract interface: accept messages from application contract after executing necessary business logic
Sender contract		Functional	sender-contract-2	Emit an event with message details, transaction hash, and metadatfor attester
Sender contract		Functional	sender-contract-3	Store message and associated datuntil successfully processed by attester and relayer
Sender contract		Functional	sender-contract-4	Ensure only authorized application contracts can send messages
Sender contract		Functional	sender-contract-5	Re-emit events if attester does not respond within certain timeframe
Sender contract		Non-Functional	sender-contract-6	Implement security features such as message nonce (replay attack)
Sender contract		Non-Functional	sender-contract-7	Optimize event emission and storage for gas efficiency
Attester	As developer, I want message transmitted from source to destination chain to be trusted, e.g. I can verify it's validity.	Functional	attester-1	Listen continuously for events emitted by sender contract
Attester		Functional	attester-2	Validate that message is correctly included in block by generating merkle proof
Attester		Functional	attester-3	Create and sign merkle proof attesting to validity of message and its inclusion in block
Attester		Functional	attester-4	Operate full node of source chain for access to block data
Attester		Non-Functional	attester-5	Ensure attestation process is secure and prevents tampering or forgery
Attester		Non-Functional	attester-6	Handle high volume of messages without significant delays (scalability)
Relayer	As developer, I want crosschain messaging protocol to be modular, such that independent nodes can focus on transmition only, and another can focus on attestation.	Functional	relayer-1	Transmit signed attestation, along with message and merkle proof, to receiver contract on destination chain
Relayer		Functional	relayer-2	Ensure datintegrity during transmission, preventing alteration or loss
Relayer		Functional	relayer-3	Implement reliable mechanisms for guaranteed message delivery, including redundancy
Relayer		Non-Functional	relayer-4	Operate under an incentivization mechanism to ensure consistent and honest participation
Relayer		Non-Functional	relayer-5	Be resilient against network issues, potential attacks, and operational challenges (fault tolerance)
Receiver contract	As developer, I want message on destination chain to be verified, e.g. by validating merkle proofs and comparing block headers, to ensure authenticity of messages.	Functional	receiver-contract-1	Verify merkle proof provided by relayer by recalculating merkle root
Receiver contract		Functional	receiver-contract-2	Validate block header by checking consistency against datstored in block header storage contract
Receiver contract		Functional	receiver-contract-3	Verify attestation’s signature to ensure it was created by trusted attester
Receiver contract		Functional	receiver-contract-4	Ensure that message, merkle proof, and block header correspond correctly
Receiver contract		Non-Functional	receiver-contract-5	Implement security measures to prevent malicious or fraudulent submissions
Receiver contract		Non-Functional	receiver-contract-6	Optimize contract operations for gas efficiency, particularly in proof verification and signature checking
Block header oracle	As developer, I want provision of block headers for checking validity of message to be secure, accurate and fast.	Functional	blockheader-oracle-1	Retrieve and verify block headers from source chain continuously using light client
Block header oracle		Functional	blockheader-oracle-2	Ensure timely updates of block headers, particularly after blocks reach finality
Block header oracle		Non-Functional	blockheader-oracle-3	Operate as decentralized oracle network to prevent single points of failure or manipulation
Block header oracle		Non-Functional	blockheader-oracle-4	Implement consensus mechanism among oracle nodes to agree on correctness of block headers
Block header oracle		Non-Functional	blockheader-oracle-5	Securely transmit block headers to destination chain to prevent tampering or interception during transmission
Block header storage contract	As developer, I want to be able to reliably provide trustable block headers to receiver contract, such that message can be validated.	Functional	blockheader-contract-1	Store verified block headers from source chain once received from oracle network
Block header storage contract		Functional	blockheader-contract-2	Only store block headers that have been confirmed as final according to source chain’s consensus rules
Block header storage contract		Functional	blockheader-contract-3	Allow only authorized contracts (e.g. receiver contract) to query stored block headers
Block header storage contract		Non-Functional	blockheader-contract-4	Ensure stored block headers are immutable and cannot be altered once stored
Block header storage contract		Non-Functional	blockheader-contract-5	Optimize storage and retrieval operations to minimize gas costs while ensuring rapid access to block headers
Block header storage contract		Non-Functional	blockheader-contract-6	Maintain history of block headers for auditability, allowing traceability and verification of past events

8.3. GitHub project

See GitHub project with further resources: https://github.com/azommerfelds/interop-platform.

9. Further reading

- Articles on bridge evaluation frameworks:
  - Crosschain Risk Framework by ConsenSys & LI.FI
  - L2 Beat Bridge Risk Framework by L2Beat
  - Assessing Blockchain Bridges by Joel John
  - With Bridges, Trust Is a Spectrum: A Quantified Framework by LI.FI and CryptoStats
- Article: Uniswap Bridge Assessment Report
- Article: ERC7683: The Cross-Chain Intents Standard
- Article: A standard interface for arbitrary message bridges between chains/layers
- Documentation of message passing solutions:
  - CCIP docs
  - zkBridge docs
  - Wormhole docs
  - LayerZero docs
  - Across docs
  - Hashi docs
- Presentation: On evaluation of crosschain messaging protocols
- Presentation: On crosschain bridge protocol design
- Paper: Cross-Blockchain Communication Using Oracles With an Off-Chain Aggregation Mechanism Based on zk-SNARKs
- Paper: A Comprehensive Overview of Security Vulnerability Penetration Methods in Blockchain Cross-Chain Bridges
- Paper: A Brief History of Blockchain Interoperability
- Paper: zkBridge: Trustless Cross-chain Bridges Made Practical
- Paper: Cross-Blockchain Communication Using Oracles With an Off-Chain Aggregation Mechanism Based on zk-SNARKs