ChatGPT

Building a Substrate-to-Base Bridge for Token Migration

Overview and Rationale

When launching a new Substrate-based chain as a fork of an old chain, a cross-chain bridge is essential to allow holders of the old chain's native token to migrate or "exit" their holdings. In this scenario, the plan is to bridge tokens **through Base** (Coinbase's Ethereum L2 network) as an intermediary. This means the old chain's token would be locked and represented on Base, and from Base it can be transferred to the new Substrate chain's native token. Using Base as a hub has two advantages: (1) it provides an accessible **exit route** for users not joining the new chain (since Base connects to the broader Ethereum ecosystem for liquidity), and (2) it leverages existing Ethereum-compatible infrastructure for the bridge's smart contracts.

Key requirements: The bridge should be **permissionless and decentralized**, ideally implemented via smart contracts and trust-minimized relayers rather than a single custodian. Holders must be able to move tokens from the old chain to the new chain (via Base) freely. A return path from Base *back to the old chain* is optional (it can be omitted to simplify design if the old chain will be deprecated). However, a **two-way bridge between Base and the new chain** is desirable so that users can move assets into and out of the new chain as needed.

Lock-Mint Burn-Release: How Asset Bridges Work

At a high level, bridging a token across chains uses a **lock-and-mint** (and the reverse burn-and-release) mechanism:

- **Lock on Source, Mint on Target:** The original tokens are **locked** or escrowed on the source chain, and an equivalent amount of **wrapped or synthetic tokens** is **minted** on the destination chain [1] [2]. The minted token represents the original asset and can circulate on the destination chain. For example, projects like Centrifuge locked their native Substrate tokens and issued ERC-20 representations on Ethereum using this approach [3]. Similarly, ChainBridge (a bridging framework) allowed Ethereum Classic users to lock up ETC on its chain and mint "synthetic ETC" on Ethereum for use in DeFi [4].

- **Burn on Source, Release on Target (Reverse Path):** If a two-way bridge is implemented, the reverse involves **burning or locking** the token on the current chain and **releasing** the equivalent tokens on the other chain. In the lock-mint model, burning the wrapped token on the target chain proves the user has redeemed it, which triggers unlocking the original token on the source chain back to the user. In our case, one direction (old chain → Base → new chain) uses lock & mint, while the reverse (new chain → Base, and optionally Base → old chain) would use burn & release.

This scheme ensures the total circulating supply remains constant. As one source describes, transferring tokens cross-chain involves burning them on the origin and minting on the destination, keeping supply global and stable [5] [6]. In a **mapping token** design, *whoever holds the issued token on the target chain can always claim the locked token on the source* [1]. In other words, the bridge creates a 1:1 backed representation: the locked "backing" tokens guarantee the value of the issued tokens [1].

Bridging via Base as an Intermediary

Using Base as an intermediary means you will effectively set up **two bridge connections**:

1. **Old Chain → Base:** Users who want to leave the old chain (not migrating) can lock their native tokens on the old chain and receive an equivalent ERC-20 on Base. This gives them the freedom to, say, swap those tokens on Base's DeFi or bridge further to Ethereum mainnet (allowing an "exit" from the old economy). This leg can be implemented as a one-way bridge (no need for Base → old chain if we assume the old chain won't receive new tokens).

2. **Base → New Chain (and vice versa):** Users who want to join the new chain will take the token on Base and move it into the new Substrate chain. They would lock or burn the token on Base and mint the new chain's native token. Likewise, in the future, new-chain users may want to bridge out to Base (for liquidity or to exit the new economy), so the **new chain ↔ Base bridge should be two-way**. Essentially, Base will hold the "canonical" locked reserve of tokens migrating between the two economies.

By routing through Base, a user could theoretically perform a two-hop migration (old→Base, then Base→new). In practice, you can streamline this with a UI or scripts, but under the hood it's two sequential bridge operations. Using an intermediate chain like this is a known strategy to connect disparate ecosystems (e.g., bridging via Ethereum as a common denominator) [7] [8]. It avoids building a direct custom bridge between the old and new chain, instead leveraging the well-supported Base/Ethereum environment in between.

Bridge Implementation Options

There are a few approaches to implement the bridging mechanism, balancing ease of development, security, and decentralization:

1. Using ChainBridge (Substrate ↔ Ethereum Bridge Pallet)

One readily available solution is ChainSafe's **ChainBridge**, which is a modular bridge framework supporting Substrate-based chains and EVM chains [3]. ChainBridge has been used to connect Substrate chains (like Centrifuge or Meter) with Ethereum and other EVM networks [3].

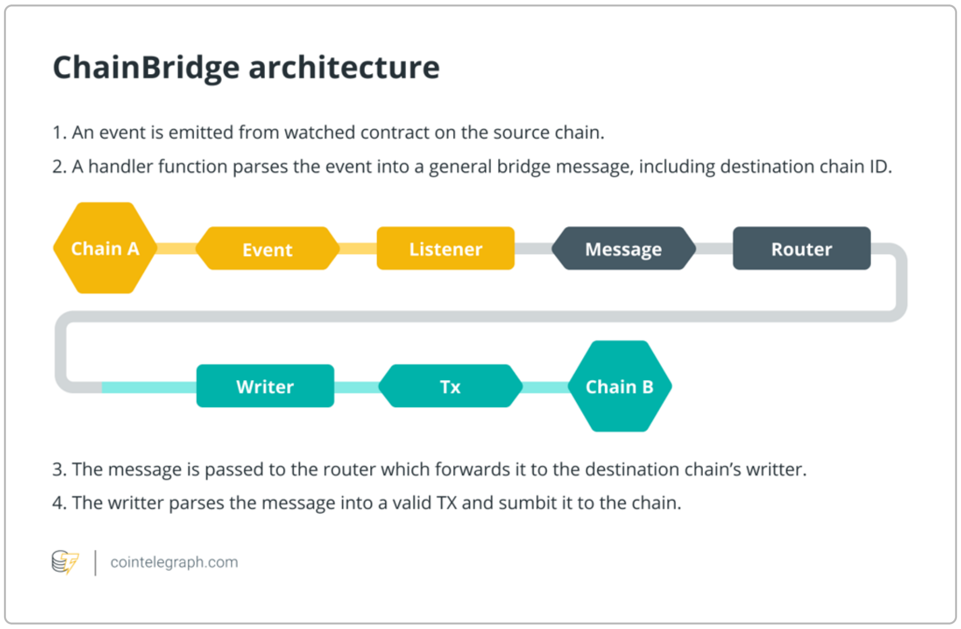

ChainBridge architecture
1. An event is emitted from watched contract on the source chain.
2. A handler function parses the event into a general bridge message, including destination chain ID.
Chain A
Event
Listener
Message
Router
Writer
Tx
Chain B
3. The message is passed to the router which forwards it to the destination chain's writter.
4. The writter parses the message into a valid TX and sumbit it to the chain.
cointelegraph.com

Figure: ChainBridge architecture overview. Events on Chain A (source) are picked up by off-chain relayers, formatted into cross-chain messages, and submitted as transactions on Chain B (target). A multi-signature relayer consensus is used to authorize each transfer [9] .

How it works: On the Substrate side, you would include the ChainBridge **pallet** in your runtime. On Base (the EVM side), you deploy ChainBridge's set of **bridge smart contracts** (a main Bridge contract plus handler contracts for ERC-20, etc.). The process is as follows:

- A user on the source chain calls an extrinsic (e.g., a pallet function) to **lock** their tokens in a bridge reserve account (the ChainBridge pallet will hold them) and specify a recipient address on the target chain. This emits a bridge event on the source chain [10] .

- A set of **off-chain relayer nodes** monitors the source chain for bridge events. When one is detected, the relayers submit a proposal to the Bridge contract on Base with the details (amount, recipient, resource ID identifying the asset, etc.). ChainBridge uses a **multisignature voting** scheme: multiple relayers must vote to approve the transfer until a threshold is reached [9] . This prevents a single relayer from hijacking funds – the design requires a quorum to authorize a mint/release [9] .

- Once enough votes are gathered, the Bridge contract calls the ERC-20 **handler** contract to mint the specified amount of the token to the recipient's address on Base [11] [4] . (If the token is being bridged *from* Base to the Substrate chain, the process is reversed: the user deposits tokens into the Base contract, and relayers instruct the Substrate pallet to unlock tokens to the target account).

In ChainBridge's model, the ERC-20 on Base can be a **mintable representation** of the Substrate token. For example, in the Ethereum Classic bridge, locking ETC on the original chain led to minting "Synthetic ETC" on Ethereum [4] . In your case, you might deploy an ERC-20 contract on Base representing the old chain's token (let's call it "OLDWrapped") which is minted when old tokens are locked. Similarly, for the new chain, you could have a "NEWWrapped" ERC-20 on Base that corresponds to new chain tokens (or reuse the same token contract if you intend it to represent the unified asset).

Decentralization: This approach is **permissionless for users** (anyone can use the bridge contracts), but you do rely on a set of relayer nodes. To decentralize this, you can run multiple independent relayers (possibly run by different community members or organizations) and use a high threshold for approvals [9]. No single party can unilaterally mint or release tokens, since the Bridge contract will only execute a transfer when enough signatures are present. This is a **semi-trusted (federated) model** – more decentralized than a single custodian, though not fully trustless. The ChainBridge team has outlined plans to move toward more trustless, light-client-based verification in future versions [12], but the current model is a well-tested solution to start with.

Development effort: ChainBridge provides most of the components off-the-shelf. You would need to integrate the pallet in both the old and new Substrate chain runtimes (configuring the bridge pallet with the appropriate resource IDs for your token, and setting up governance or sudo to manage relayer registration and perhaps emergency pauses). On Base, you'd deploy the standard ChainBridge contracts (there are CLI tools like `cb-sol-cli` to help with deployment and configuration of resource IDs, handler contracts, etc. [13] [14]). Finally, you'll operate the relayer processes (which can be run via Docker as provided by ChainSafe [15]) and have them connected to both chains' RPC endpoints.

2. Using Cross-Chain Message Protocols (LayerZero, etc.)

Another approach is to leverage emerging **cross-chain messaging networks** like **LayerZero** to handle the heavy lifting of message delivery between chains. LayerZero provides a generic messaging protocol and has an **omnichain token standard (OFT)** that extends a token across multiple chains with a unified supply [5] [6]. Projects can deploy their token as an OFT so that users can move it between any LayerZero-supported chains.

LayerZero **already supports Base** network, and it has been integrated into some Substrate ecosystems (for example, Astar Network uses LayerZero to transfer assets between its Polkadot parachain and an EVM chain) [16]. The pattern with LayerZero's OFT is a **burn-and-mint** mechanism: when moving from Chain A to Chain B, the token is burned on A and a message is sent to mint it on B [5]. This is analogous to lock/mint, except that if the token exists on multiple chains, they treat one chain as the source of truth for minting or use a total supply across chains. In your case, since the token originates on the old chain, you could deploy an **OFT Adapter** contract on Base that locks the old token representation and triggers minting on the new chain [17] (and vice versa for return trips).

How it might work in practice:

- You would deploy a LayerZero endpoint (smart contract) on Base along with an OFT token contract (or adapter) for your asset. On the new Substrate chain, if it has an EVM or smart-contract capability (e.g., via an embedded EVM pallet or an ink! contract), you'd deploy a corresponding endpoint and token contract there as well. When a user on Base invokes a transfer to the new chain, the LayerZero contract packages a message with the amount and recipient. An off-chain **LayerZero Relayer** and **Oracle** then deliver proof of this message to the new chain's contract, which verifies it and mints the tokens to the recipient. LayerZero's default security model relies on two independent parties (an oracle and a relayer) to attest to the message; if at least one is honest, the message is valid – this avoids a single point of trust.

- LayerZero's approach is **permissionless** from a user perspective and does not require you to run your own bridge network, but you are **trusting the LayerZero network** and its chosen oracles/relayers (often Chainlink or similar) for message delivery. The upside is a very seamless user experience and compatibility with many chains. For example, the Astar team notes that LayerZero's Stargate bridge support enabled easy transfers between Astar's Substrate chain and its EVM chain [16].

Using LayerZero for your case would require that your new chain is made LayerZero-compatible. If the new chain is a Polkadot parachain or Substrate solo chain without EVM, direct integration is non-trivial – you might need a custom LayerZero endpoint pallet. (Some networks like Moonbeam and Astar achieved this by leveraging their built-in EVM environments for the LayerZero contracts, or by collaboration with LayerZero Labs). This option is worth considering if you plan for long-term cross-chain interoperability and want to plug into a broader omnichain ecosystem. It gives you out-of-the-box support for bridging to many chains (not just Base) using the same token standard [18] [5]. However, it may involve more upfront development/integration work if not already supported by your chain.

3. Custom Light-Client Bridge (Trustless Bridging)

For completeness, there is a fully trustless design approach: implementing on-chain **light clients** so that each chain can verify the other's state transitions. This is the approach of projects like **Snowbridge (Snowfork)** for Polkadot ↔ Ethereum and Darwinia's bridges. A light-client bridge means embedding the verification logic of Chain A into Chain B (and vice versa) so that no external validators are needed – the target chain itself checks cryptographic proofs of the source chain's block finality and transactions [19]. In such a design, a relayer only needs to forward raw data (headers and proofs), and the smart contract or pallet on the receiving side will validate it against the embedded light client.

While this is the most secure (no trust in off-chain agents beyond honest majority assumptions of each chain), it is **complex and expensive** to implement. For instance, Snowbridge's Ethereum contract must verify Polkadot/Kusama finality proofs and vice versa, which is a heavy computation (and indeed, Snowfork is still under active development and not yet production-ready) [20] [21]. Light client bridges also incur high on-chain fees to verify proofs (especially on EVM side). Given the time and complexity, this approach is usually only pursued for very high-value permanent bridges. It likely exceeds the scope of a quick token migration bridge, but it's something that could be a **future goal** once the new chain is mature (perhaps collaborating with Polkadot's Bridge Hub or similar initiatives).

Summary of Options: For a practical implementation **now**, using a **ChainBridge-like solution** (option 1) is often the quickest path for Substrate↔EVM bridging that you control. It provides a proven lock/mint framework [10] with the ability to decentralize via a relayer set. Integrating **LayerZero or another message network** (option 2) is also viable and can simplify user experience, but ensure your new chain can support it and be mindful of the trust assumptions (relying on the LayerZero infrastructure). Fully trustless bridges (option 3) are the gold standard but involve significant development.

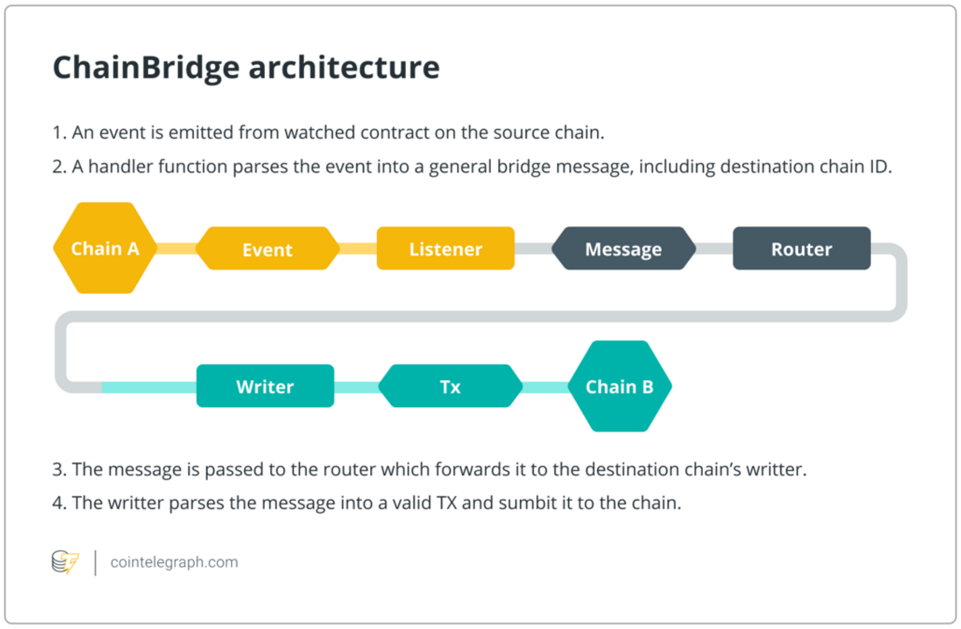
Implementation Steps

Regardless of the chosen approach, the bridge will consist of on-chain modules plus off-chain components. Here's a breakdown of the steps to build the bridge using a ChainBridge-style design (which is analogous in many ways to any lock/mint bridge):

1. Smart Contracts on Base (Ethereum L2): Develop or deploy the necessary contracts on Base. This typically includes: - An **ERC-20 token contract** for the wrapped asset (if one does not already exist). This token will represent the old chain's token on Base. (If you plan for the same representation to be used for bridging into the new chain, you might only need one ERC-20 that serves as the intermediary asset on Base). - A **Bridge contract** that orchestrates cross-chain transfers. ChainBridge, for example, has a generic Bridge contract that accepts proposals from relayers. - **Handler contracts** for each asset type (e.g., an ERC20 handler that knows how to mint/burn the ERC-20). The handler is called by the Bridge contract when a proposal is executed. In a custom implementation, the Bridge logic and token minting logic might be combined, but following a modular approach is safer. - These contracts should be deployed with appropriate initial parameters (setting your new chain and old chain identifiers, the token address, etc.). In ChainBridge, you register a resource ID that ties together the source asset and the target handler [22] [23] .

2. Substrate Pallet on Old and New Chains: On the Substrate side, you need runtime logic to lock and release tokens: - For the **old chain**, the pallet should have a function (callable by users) to **lock** a specified amount of native tokens and record a pending outbound transfer. The pallet would emit an event with details (amount, target chain = Base, target recipient address). In ChainBridge's pallet, for example, there is a `transfer` call that does this, and you configure the pallet with the resource ID of the token and the target chain ID [24] . - The pallet also needs to handle **inbound transfers** (if you allow Base→old, though you said that might not be needed). Inbound means reacting to a relayer instruction to unlock tokens. Usually, the bridge pallet will have an extrinsic (originating from the relayer, perhaps with root or an assigned role) to execute a **release** of tokens to a user if a valid cross-chain message is observed. - On the **new chain**, you'll have a similar pallet. It must be able to **mint** new native tokens when it receives an inbound transfer from Base (since users moving to the new chain need to get newly issued tokens there). You might design the new chain's token supply to be inflationary so the pallet can mint against the locked reserve on Base. Conversely, for outbound new→Base, the pallet would burn or lock the user's tokens and signal the event to Base. - Both pallets need admin configuration: e.g., setting the **Bridge account** (an account that holds locked funds), registering **relayer identities or threshold multisig** if using ChainBridge, and mapping resource IDs to methods (as seen in ChainBridge's setup where they link a 32-byte resource ID with a function call like `Example.transfer` on the pallet) [25] .

3. Relayer (Off-chain) Setup: If using your own bridge (ChainBridge or custom), set up a **relayer network**: - Determine how many relayers and who will run them. For decentralization, enlist multiple parties. Each relayer will need keys for submitting transactions on Base and on the Substrate chains. You can use a threshold multi-signature scheme or have the Bridge contract require M-of-N signatures (ChainBridge handles this internally by each relayer's vote). - Configure the relayer software with the chain endpoints and contract addresses. For example, ChainBridge's config will include the RPC URL for Base, the Bridge contract and handler addresses, and the WebSocket for the Substrate node with the pallet name, etc. [26] [27] . - Relayers listen for events on each side. On the Substrate side, they'll watch for the pallet's `Transfer` (or similar) event for outgoing transfers. On Base's side, they'll watch for the Bridge contract's deposit events (if handling Base→Substrate). - When an event is seen, the relayer composes the cross-chain **message** and submits it to the opposite chain. In ChainBridge's architecture (see **Figure** above), the sequence is: Event on Chain A → off-chain Listener → message relayed to Chain B's Router/Handler → target chain executes the transaction


ChainBridge architecture
1. An event is emitted from watched contract on the source chain.
2. A handler function parses the event into a general bridge message, including destination chain ID.
Chain A
Event
Listener
Message
Router
Writer
Tx
Chain B
3. The message is passed to the router which forwards it to the destination chain's writter.
4. The writter parses the message into a valid TX and sumbit it to the chain.
cointelegraph.com

[11] . - **Security:** Ensure relayer keys are secure. If using a multisig approach, no single relayer's key can move funds; they only propose and vote on transfers which the contract finalizes after quorum. Still, you should vet relayer operators and perhaps require bonding or stake (some bridges have used incentive mechanisms so relayers behave honestly).

4. Testing the Bridge: Deploy everything on a testnet or local setup first. Test scenarios: - Lock on old chain → Mint on Base. Verify that the correct amount of ERC20 appears in the user's address on Base after finality. - Burn on Base → Mint on new chain. This tests the new chain's ability to receive messages. You might have to simulate this if your new chain isn't live yet; a local Substrate node running the pallet could be used in tandem with a local Ethereum dev network. - Try invalid or edge cases (e.g., lock 0 tokens, or relayer fails to relay) to ensure system handles them (funds should remain locked if not processed, etc.). - **No Base→Old path:** Since you anticipate not needing Base back to old chain, configure the system to reject or simply not process that direction. This could mean not deploying any mint handler for old chain on Base, or not whitelisting Base as a source chain in the old chain's pallet config [28] . That way, even if someone tried to send tokens from Base back to old, the message would be ignored. This effectively makes it a one-way burn for the old token (which might be fine if the old chain is being sunset).

5. Deploy and Monitor: Once tested, deploy the contracts to Base mainnet and upgrade the runtimes of the old and new chains to include the bridge pallets. Start the relayer nodes connecting the live networks. It's prudent to **monitor** the bridge activity closely initially – you can build a simple dashboard or use logs from relayers to track transfers. Additionally, impose conservative limits at first (you can cap maximum transfer per transaction or require manual approval for very large transfers) to mitigate risk while the system is new.

Security and Decentralization Considerations

Building a bridge comes with significant responsibility — historically, many bridge hacks have occurred, often due to vulnerabilities in smart contracts or weaknesses in the multi-sig/validator setup. Here are some best practices and considerations:

- **Auditing:** Thoroughly audit the smart contracts (or use well-audited ones). If using ChainBridge's contracts and pallet, review any recent security updates from ChainSafe. Similarly, audit any custom pallet code you write for locking/minting on the Substrate side. Bridges often manage large amounts of value (a "honeypot"), so they become prime targets [29].

- **Relayer Trust Model:** If you go with a federated relayer model, choose a **robust threshold** (e.g., 4-out-of-5 multisig) so that it's hard for collusion to occur. Ensure that the set of relayers is geographically and politically decentralized (different teams or community members) to reduce the chance of all being compromised simultaneously. You can also require bonded stake for relayers as an economic security measure (though ChainBridge's base design doesn't include staking, you could implement slashable offense if a relayer misbehaves by transaction censorship or otherwise).

- **Upgrade Path:** Design the contracts/pallet with upgradability or emergency pause switches. For instance, a governance-controlled flag to pause transfers if a vulnerability is discovered can save funds from being exploited. Multisig bridges are custodial, so an exploit or key compromise could mean loss of the locked funds. Having an admin ability (even if time-locked or requiring a super-majority of a DAO) to halt transfers is a safety switch – but balance this with decentralization (you don't want a single admin able to abuse the bridge either).

- **Permissionless Usage:** Make sure that once deployed and configured, anyone can use the bridge (deposit/lock their tokens without needing further approval). Your design already leans toward that by using contracts and pallets. Avoid whitelisting of users – only the relayers might be whitelisted. If you use ChainBridge, you'll use the pallet's Sudo (or governance) to add approved relayer accounts [30] and whitelist chain IDs [28], but after that, regular users interact trustlessly.

- **Future Integration:** By bridging through Base, you also open up the possibility to integrate with existing cross-chain services. For example, once the token is on Base as an ERC-20, users could leverage **third-party bridges** (like Axelar, Wormhole, Celer cBridge, etc.) to move from Base to other chains if needed. In the Polkadot ecosystem, many bridges operate via Moonbeam or similar EVM parachains [31]. If your token is on Base, those general bridges can potentially carry it further. This isn't something you need to build, but it's a side benefit to mention: bridging via a popular hub increases the token's accessibility.

- **Layer0 vs. Roll-your-own:** If using a network like LayerZero, note that you are outsourcing some security to their system. LayerZero has a good track record so far and has been used across many chains [32], but it's still a relatively new paradigm. Ensure you understand the **Security Stack** configuration – you can customize who the oracles and relayers are for your application if you want more control [33] [34]. The nice aspect of LayerZero is that your token contract remains **non-custodial** (no third party holds tokens; they are burned/minted under your contract's control) [35]. This eliminates some counterparty risk compared to a custodied multi-sig bridge. Yet, the message delivery is the critical point to secure (to prevent fraudulent mint messages).

In summary, **bridging your Substrate token via Base is feasible with today's technology**. The most straightforward route is to deploy a bridge using a **lock and mint model** with a Substrate pallet and Ethereum contracts you control. This has been demonstrated by multiple projects (locking assets on a Substrate chain and minting on Ethereum/Base) [10] [4]. By not relying on a single centralized bridge operator (since you'll have a quorum of relayers or a decentralized network service), you achieve a level of decentralization and permissionless usage. Users of the old chain get a trust-minimized way to exit: they can swap their locked tokens for a Base representation and trade freely. Users coming into the new chain likewise have a path to join by acquiring the token on Base and then bridging it in.

Conclusion

Setting up a Substrate ↔ Base bridge for your token migration is a multi-step process, but it leverages well-understood cross-chain techniques. You will be **locking the old chain's tokens and issuing a mirrored token** on Base [1] [2], then allowing that token to flow into the new chain's native currency. Whether you choose an established framework like ChainBridge or a modern messaging protocol like LayerZero, the core idea remains the same: maintain a **1:1 backing** so that holders can always redeem or migrate their value between chains. By routing through Base, you provide an elegant opt-out for those not joining the new chain (they can stay on an Ethereum-based network or cash out) and an on-ramp for those who do join (they can bring their assets over in a permissionless way).

The bridge will require careful planning and security measures, but it **already has precedent in the wild** – for example, projects have used similar bridges to move ERC-20 tokens to Substrate chains and vice versa [3] [4]. As you implement it, test thoroughly, employ community trust (multiple relayers or validators), and keep an eye on future improvements (like trustless light clients) as longer-term upgrades. With this bridge in place, your new chain's launch can accommodate both migrating users and those exiting, ensuring a fair and flexible path forward for the token holders in your ecosystem.

Sources:

- Cross-chain bridge fundamentals (lock/mint mechanism) [1] [2]
- ChainBridge design and usage between Substrate and Ethereum [3] [4]
- ChainBridge security model (relayer voting threshold) [9] and example of token transfer (ETC to Ethereum) [4]
- LayerZero omnichain token standard (OFT) and burn/mint process [5] [6]; Astar Network using LayerZero for Substrate↔EVM interoperability [16]
- Bridge implementation notes (ChainBridge setup, pallet and contract actions) [10] [24]
- Trustless bridge concept using light clients (Darwinia, Snowbridge) [19] [20], and risks of custodial bridges [29].

---

[1] [19] An Introduction to Substrate-to-Substrate Bridges | by Darwinia | RingDAO | Medium
https://medium.com/ringdao/an-introduction-to-substrate-to-substrate-bridges-7b77dc53506

[2] [16] astar.network
https://astar.network/blog/layerzero-to-provide-cross-chain-interoperability-for-astar-94037

[3] [20] [21] [29] The Polkadot architecture and introduction to the Substrate infrastructure
https://cointelegraph.com/learn/articles/the-polkadot-architecture-and-introduction-to-the-substrate-infrastructure

[4] [9] [11] [12] ChainSafe Building ChainBridge
https://blog.chainsafe.io/chainsafe-building-chainbridge/

[5] [6] [17] [18] [32] [33] [34] [35] Explaining the OFT Standard. OFTs are tokens that move across... | by Mark Murdock | LayerZero Official | Medium
https://medium.com/layerzero-official/explaining-the-oft-standard-310de5e84052

[7] [8] [31] RFC: Polkadot Bridges Explorer & Cross-Chain Intelligence - Tech Talk - Polkadot Forum
https://forum.polkadot.network/t/rfc-polkadot-bridges-explorer-cross-chain-intelligence/11182

[10] [13] [14] [15] [22] [23] [24] [25] [26] [27] [28] [30] Blockchain Interoperability Solution: How Chainbridge Can Be A Way Out?
https://www.talentica.com/blogs/blockchain-interoperability-solution-how-chainbridge-can-be-a-way-out/