ChatGPT

Sovereign Avalanche L1 – MVP Build Guide

Goal: Launch a permissioned Avalanche Layer-1 (custom subnet) on Fuji testnet that enforces strict compliance rules for an AED stablecoin and demonstrates a cross-chain payment via Avalanche's Warp Messaging (ICM) protocol [1] [2].

🏷 Phase 1: Foundation (Days 1–4) – Set Up the Blockchain

- **Create the Avalanche L1 via CLI:** Use the Avalanche-CLI to build your chain config [1]. For example, run:

```
avalanche blockchain create SovereignL1
```

  When prompted, **choose** `Subnet-EVM` as the virtual machine [3] and **select** `Proof Of Authority` **(PoA)** for validator management [4]. This creates a genesis configuration tailored for a permissioned EVM chain.
- **Set Chain Parameters:** In the CLI wizard, pick **"defaults for a test environment"** (Fuji) and enter a **unique ChainID** (e.g. any unused number ≠ 1, 43114) [5]. Specify a token symbol for the native asset (e.g. "AEDX") – this is mainly for genesis allocations [6]. After completion, export the genesis config file:

```
avalanche blockchain describe SovereignL1 --genesis > genesis.json
```

- **Configure Gas Fees:** In the `genesis.json` under `"config"`, enable fee delegation if desired. For example, setting `"allowFeeRecipients": true` lets your PoA validators collect fees rather than burning them [7]. This ensures transaction fees can be paid out to validators. *(Note: Avalanche L1s can decide how to handle fees – e.g. burn or reward validators [8].)*

- **Add Governance Keys:** After creating the chain, assign your two P-Chain addresses (Core wallet keys) as control keys. Use `avalanche blockchain changeOwner` to set both addresses with a threshold of 2 signatures. For example:

```
avalanche blockchain changeOwner SovereignL1 --control-keys <P-
addr-1>,<P-addr-2> --threshold 2
```

  This makes both core addresses joint controllers of the ValidatorManager contract. *(Do not use the public EwoQ test key in production [9].)*

- **Launch a Validator Node:** Provision a cloud VM (e.g. Azure, AWS) to run `avalanchego`. Use Ubuntu 20.04+ and allocate at least **8 vCPUs, 16 GB RAM, and 1 TB SSD** [10]. Install the AvalancheGo binary (from GitHub or apt package). Place the exported `genesis.json` and node config on the VM, then start the node pointing to your L1. For example:

```
avalanchego --genesis-file=/path/to/genesis.json --config-file=/path/to/
node-config.json
```

This boots your L1. Your node will produce blocks in PoA mode (since you are the only validator for MVP).

- **Deploy Identity Registry:** Using a framework like Hardhat or Foundry, compile and deploy an `IdentityRegistry.sol` contract to your new chain. This contract should implement role-based access (e.g. via OpenZeppelin's AccessControl) with roles `REGULATOR`, `ISSUER_STABLECOIN`, and `PARTICIPANT`. Deploy it in genesis or as the first transaction on-chain. Assign your Regulator address (one of the core addresses) the `REGULATOR` role. This on-chain registry will govern all identity/role checks.

🏷 Phase 2: Core Economics (Days 5–8) – Token and Fee Token

- **Deploy E-AED Gas Token:** Write an ERC-20 contract `E-AED.sol` (1:1 peg to AED). Deploy it on your L1, minting an initial supply (e.g. held by Regulator). Make the `REGULATOR` role the minter. After deployment, note the E-AED contract address.

- **Configure Gas Token:** Modify your L1's chain config so that **E-AED is used for gas fees**. In practice, edit `chainConfig` (genesis or via `avalanche blockchain configure`) to set the gas token address to your E-AED contract. (Avalanche's Subnet-EVM supports specifying a custom gas token; refer to the Subnet-EVM docs for the exact key.) Also set `"gasPrice":1` or similar in genesis so that gas costs are paid in E-AED. Because `allowFeeRecipients` is enabled, validators will receive E-AED fees to their chosen address [7]. *(This replaces the default AVAX model.)*

- **Implement Gas Fee Logic:** In your L1 contracts, charge gas in E-AED. The chain will automatically deduct fees in E-AED if configured. You may also include a simple distribution mechanism: for example, set the block reward or fee split to send a portion of gas fees to the producer's address.

- **Deploy AED Stablecoin Contract:** Develop `AEDStablecoin.sol` that enforces compliance:

- Only accounts with the `ISSUER_STABLECOIN` role in the IdentityRegistry can call `mint` or `burn`.
- `transfer` should check that **both** `msg.sender` **and** `to` **have the** `PARTICIPANT` **role**; otherwise revert.
- No staking, lending, or other features – just ERC-20 value transfer.
- Integrate the registry by having the contract reference its address and calling its role-check functions.
  Deploy this contract (via Hardhat) on your L1. Test it so that minting by the Issuer works and all unauthorized actions (e.g. non-issuers trying to mint, or sending to a non-participant) revert as expected.

🏷 Phase 3: ICM Integration (Days 9–11) – Cross-Chain Messaging

- **Study Avalanche Warp Messaging:** Avalanche's Interchain Messaging (ICM) provides an EVM precompile for sending signed messages between L1s [2]. These messages carry arbitrary data and are signed by validators (via BLS) on the source chain [11] [12]. Importantly, when an L1 (your subnet) receives a message from the C-Chain, it will verify the message using *your* validator set rather than the entire primary network [13]. This makes Fuji→L1 messages efficient: only a threshold of your L1's stake needs to sign, not all of Avalanche's stake [14].

- **Deploy C-Chain Payment Processor:** On Avalanche Fuji's C-Chain, write a `PaymentProcessor.sol` contract (or use the Teleporter example). Its workflow: accept a user's funds (e.g. AVAX or stablecoin) and then call the Warp Message precompile to emit a cross-chain message. The message payload should include the recipient L1 address, the AED amount, and any reference. Use the precompiled contract (e.g. via `sendWarpMessage(...)`) so the message is signed by C-Chain validators. (See Avalanche's Warp docs for the exact ABI.)

- **Deploy L1 Bridge Manager:** On your L1, deploy `RegulatedBridgeManager.sol`. This contract should listen for incoming Warp messages (via the Warp precompile interface). In its message handler:

- **Verify Message:** Use the Warp precompile's verification function to ensure the message is authentic and came from your licensed C-Chain contract.
- **Check Permissions:** Check that the sender (the payment processor or end-user) is authorized per your IdentityRegistry (e.g. has `PARTICIPANT` role).

- **Execute Transfer:** If valid, call your `AEDStablecoin` contract to transfer/mint the AED to the recipient's L1 address.
  (For simplicity, assume one-way payments for MVP; no funds are sent back on C-Chain except via a separate redemption call or relayer acknowledgement.)

- **Demo Cross-Chain Flow:** Test the entire flow: a user on C-Chain locks funds and calls `PaymentProcessor`. That emits a Warp message. After a short delay, your L1's bridge contract receives and verifies it, then executes the AED transfer on L1. Finally, have the C-Chain contract release funds to the beneficiary (this can be manual or via an ICM acknowledgment). This flow demonstrates a **compliant UAE→India payment**: C-Chain (Fuji) handles the fiat on-ramp, and your L1 enforces all compliance (via IdentityRegistry checks) when crediting the AED to the Indian participant.

🏷 Phase 4: Testing & Finalization (Days 12–14)

- **Write Comprehensive Tests:** Using Hardhat or Foundry, implement unit and integration tests for all contracts. Include tests for:
- **IdentityRegistry:** role grants/revokes.
- **E-AED token:** minting, gas payments (optionally simulate a simple transaction to see gas deduction).
- **Stablecoin compliance:** unauthorized mint or transfer attempts should revert. Authorized mint/transfer should succeed.
- **ICM flow:** Simulate Warp messaging in tests. (You can call the Warp precompile directly in a Hardhat test to mimic C-Chain signatures.) Verify that unverified or tampered messages are rejected.

- **Validator slashing (simulated):** Optionally test that if a validator misbehaves (e.g. you could simulate by voting against a valid message), the message is not considered valid under the threshold policy.
  All tests should pass for valid scenarios and explicitly *fail* for any compliance violation by design.

- **Basic Admin/Monitoring Interface:** Provide simple scripts or a web dashboard for chain administration:

- **Validator status:** Use AvalancheGo's APIs (or console) to monitor the node's health and block production.
- **Role management UI:** A form to call IdentityRegistry's grant/revoke functions for roles (only accessible to the Regulator).
- **Transaction logs:** Display recent transactions involving the stablecoin or bridge (via RPC queries).

- **ICM test utility:** A button to simulate sending a test cross-chain payment.
  (This UI can be minimal – even a static HTML page with Web3 connectivity.)

- **Documentation and Demo Prep:** Assemble all deliverables: smart contract code, deployment scripts, modified genesis, and a detailed README. Record a demo video showing:

- L1 node booting (via console).
- Deployment of contracts (via Hardhat).
- A sample compliant flow: user initiates on C-Chain, funds lock, L1 executes AED transfer.
  Annotate the video to highlight compliance checks (e.g. show a failed transfer when roles are missing).

MVP Checklist

By the end of Day 14, you should have:

- A **running Avalanche L1** on Fuji (single PoA validator) created with Subnet-EVM [3] [4].
- IdentityRegistry deployed with `REGULATOR`, `ISSUER_STABLECOIN`, and `PARTICIPANT` roles, and initial role assignments.
- E-AED ERC-20 token configured as the chain's gas currency (via genesis config) instead of AVAX [7]. Validators receive gas fees in E-AED.
- AED stablecoin contract enforcing mint/transfer rules via IdentityRegistry checks. No unauthorized minting or transfers are possible.
- Warp Messaging set up: PaymentProcessor on C-Chain can emit messages, and RegulatedBridgeManager on L1 can verify them using the Warp precompile [15].
- A working cross-chain payment: funds locked on Fuji C-Chain trigger a compliant AED transfer on your L1, then funds are released on Fuji. All steps respect roles/allowlists.
- Tests covering all success and failure paths, ensuring compliance logic is airtight.

Key References: Official Avalanche docs and guides on Avalanche-CLI and Warp Messaging have been followed closely [1] [2] [15] [10] [7]. These will help clarify any tool-specific commands or config details as you implement each step.

---

1 3 4 5 6 9 Create Avalanche L1 | Avalanche Builder Hub
https://build.avax.network/docs/tooling/avalanche-cli/create-avalanche-l1

2 11 12 What is ICM? | Avalanche Builder Hub
https://build.avax.network/docs/cross-chain/avalanche-warp-messaging/overview

7 Customize an Avalanche L1 | Avalanche Builder Hub
https://build.avax.network/docs/avalanche-l1s/evm-configuration/customize-avalanche-l1

8 Gas Fees and Gas Limit | Avalanche Builder Hub
https://build.avax.network/academy/avalanche-l1/customizing-evm/05-genesis-configuration/04-gas-fees-and-limit

10 Microsoft Azure | Avalanche Builder Hub
https://build.avax.network/docs/nodes/run-a-node/on-third-party-services/microsoft-azure

13 14 15 Integration with EVM | Avalanche Builder Hub
https://build.avax.network/docs/cross-chain/avalanche-warp-messaging/evm-integration