Cross-Chain Bridge - BSC ↔ Ethereum

A production-grade cross-chain bridge implementation with multi-relayer consensus, built following SOLID principles and design patterns.

🏗️ Architecture Overview

System Components

```
┌─────────────┐
│  Frontend   │  React + ethers.js + Tailwind
│   (Vite)    │
└──────┬──────┘
       │
   ┌───▼────────────────────┐
   │                        │
┌──▼────────┐        ┌──────▼───────┐
│    BSC    │        │  Ethereum    │
│  Testnet  │        │   Sepolia    │
└──────┬────┘        └──────┬───────┘
       │                    │
       └────────┬───────────┘
                │
        ┌───────▼──────────┐
        │  AWS SAM Backend │
        │   Serverless     │
        │                  │
        │  ┌─────────────┐ │
        │  │ EventPoller │ │ (3 instances)
        │  ├─────────────┤ │
        │  │  Validator  │ │ (DynamoDB Streams)
        │  ├─────────────┤ │
        │  │  Executor   │ │ (Mint/Unlock)
        │  ├─────────────┤ │
        │  │     API     │ │ (REST endpoints)
        │  └─────────────┘ │
        │                  │
```

```
         │  DynamoDB Single │
         │      Table       │
         └──────────────────┘
```

Design Patterns Used

- **Singleton Pattern**: Logger, service instances
- **Factory Pattern**: Error creation, entity creation
- **Strategy Pattern**: Chain-specific pollers, consensus validation
- **Observer Pattern**: DynamoDB Streams trigger validator
- **Command Pattern**: Bridge operations (lock/mint/burn/unlock)
- **Repository Pattern**: DynamoDB service abstraction
- **Facade Pattern**: Web3Service, BridgeService
- **Template Method**: Common polling/validation flows
- **Guard Pattern**: Input validation, reentrancy protection

SOLID Principles

✅ **Single Responsibility**: Each class/function has one reason to change
✅ **Open/Closed**: Extensible without modification
✅ **Liskov Substitution**: Interfaces are properly implemented
✅ **Interface Segregation**: Small, focused interfaces
✅ **Dependency Inversion**: Depends on abstractions, not concretions

DRY (Don't Repeat Yourself)

- Shared utilities (logger, errors) used across all Lambda functions
- Centralized configuration (chains, contracts)
- Reusable React hooks (useWeb3, useBridgeContract)
- Common validation and signing logic

📦 Project Structure

```
cross-chain-bridge/
├── contracts/                 # Smart Contracts (Hardhat)
│   ├── BEP20Token.sol        # BSC: Original token (1M supply)
│   ├── WrappedToken.sol      # Ethereum: Wrapped token
│   ├── BSCBridge.sol         # BSC: Lock/Unlock
│   ├── EthereumBridge.sol    # Ethereum: Mint/Burn
│   ├── interfaces/           # IBEP20, IWrappedToken
│   ├── scripts/deploy.js     # Deployment automation
│   └── test/                 # Contract tests
│
├── backend/                   # AWS SAM Backend
│   ├── src/
│   │   ├── functions/
│   │   │   ├── eventPoller/  # Polls BSC/Ethereum events
│   │   │   ├── validator/    # Validates consensus
│   │   │   ├── executor/     # Executes mint/unlock
│   │   │   └── api/          # Status/health endpoints
│   │   └── shared/
│   │       ├── config/       # Chain & contract configs
│   │       ├── services/     # DynamoDB, Web3, Signing
│   │       └── utils/        # Logger, errors
│   ├── template.yaml         # AWS SAM infrastructure
│   └── samconfig.toml        # Deployment config
│
└── frontend/                  # React Frontend (Vite)
    ├── src/
    │   ├── components/       # UI components
    │   ├── hooks/            # useWeb3, useBridgeContract
    │   └── App.jsx           # Main app
    ├── vite.config.js
    └── tailwind.config.js
```

🚀 Deployment Guide

Prerequisites

- Node.js 20+
- AWS CLI configured
- AWS SAM CLI installed
- MetaMask wallet with testnet funds

Step 1: Deploy Smart Contracts

```
# Install dependencies
npm install

# Set environment variables
```

```
cp .env.example .env
# Edit .env with your RPC URLs and deployer private key

# Compile contracts
npm run compile

# Deploy to BSC Testnet
npm run deploy:bsc

# Deploy to Ethereum Sepolia
npm run deploy:eth

# Save contract addresses from output
```

Step 2: Create AWS Secrets

```
# Create secrets for relayer private keys
aws secretsmanager create-secret \
    --name Relayer1PrivateKey \
    --secret-string '{"privateKey":"YOUR_PRIVATE_KEY_1"}'

aws secretsmanager create-secret \
    --name Relayer2PrivateKey \
    --secret-string '{"privateKey":"YOUR_PRIVATE_KEY_2"}'

aws secretsmanager create-secret \
    --name Relayer3PrivateKey \
    --secret-string '{"privateKey":"YOUR_PRIVATE_KEY_3"}'
```

Step 3: Deploy Backend (AWS SAM)

```
cd backend

# Install dependencies
npm install

# Build SAM application
sam build

# Deploy (first time - guided)
sam deploy --guided
```

```
# Provide parameters:
# - Stage: dev
# - BSCRpcUrl: Your BSC RPC URL
# - EthereumRpcUrl: Your Ethereum RPC URL
# - Contract addresses (from Step 1)

# Subsequent deployments
sam deploy
```

Step 4: Deploy Frontend

```
cd frontend

# Install dependencies
npm install

# Set environment variables
cp .env.example .env
# Edit .env with:
# - API Gateway URL (from SAM output)
# - Contract addresses (from Step 1)

# Development
npm run dev

# Production build
npm run build

# Deploy to Vercel/Netlify
# - Connect GitHub repo
# - Add environment variables
# - Deploy
```

🧪 Testing

Smart Contract Tests

```
# Run all tests
npm test
```

```
# Run specific test file
npx hardhat test test/token.test.js

# Test with coverage
npx hardhat coverage
```

Backend Tests

```
cd backend
npm test
```

Local Development

```
# Start local SAM API
cd backend
sam local start-api --env-vars env.json

# Start frontend
cd frontend
npm run dev

# Frontend will be available at http://localhost:3000
```

📊 DynamoDB Single Table Design

Table: BridgeTable

Primary Key:

- `PK` : `EVENT#{eventId}` (Partition Key)
- `SK` : `METADATA` | `SIGNATURE#{relayerId}` | `EXECUTION` (Sort Key)

GSI1 (Query by Chain/Status):

- `GSI1PK` : `CHAIN#{chain}`
- `GSI1SK` : `STATUS#{status}#{timestamp}`

GSI2 (Query Signatures):

- `GSI2PK`: `EVENT#{eventId}`
- `GSI2SK`: `SIGNATURE#{relayerId}`

Entity Types

1. **Event** (`SK=METADATA`):
    - txHash, chain, amount, fromAddress, toAddress, status, timestamps
2. **Signature** (`SK=SIGNATURE#{relayerId}`):
    - signature, relayerId, publicKey, timestamp
3. **Execution** (`SK=EXECUTION`):
    - status, txHash, retryCount, error, timestamps

Query Patterns

```
// Get all data for an event (1 query)
Query PK=EVENT#{eventId}

// Get events by chain and status
Query GSI1 where GSI1PK=CHAIN#{chain} and GSI1SK begins_with STATUS#{stat

// Get signatures for an event
Query PK=EVENT#{eventId} and SK begins_with SIGNATURE#
```

🔐 Security Features

Smart Contracts

- ✅ OpenZeppelin ReentrancyGuard
- ✅ Ownable access control
- ✅ Pausable functionality
- ✅ Input validation
- ✅ Event deduplication

Backend

- ✅ AWS Secrets Manager for private keys

- ✅ IAM roles with least privilege
- ✅ DynamoDB encryption at rest
- ✅ Multi-relayer consensus (2-of-3)
- ✅ Signature verification

Frontend

- ✅ MetaMask integration
- ✅ Chain validation
- ✅ Transaction approval flow
- ✅ CORS protection

📈 Monitoring & Logging

CloudWatch Logs

```
# View EventPoller logs
aws logs tail /aws/lambda/dev-EventPoller --follow

# View Validator logs
aws logs tail /aws/lambda/dev-Validator --follow

# View Executor logs
aws logs tail /aws/lambda/dev-Executor --follow
```

API Endpoints

- `GET /health` - Health check
- `GET /system-info` - System status
- `GET /status?eventId={id}` - Event status
- `GET /stats` - Bridge statistics

Example:

```
curl https://your-api-gateway-url.amazonaws.com/dev/health
```

🔄 Multi-Relayer Consensus Flow

1. **Event Detection** (Every 30s):
    - EventBridge triggers 3 EventPoller Lambda instances
    - Each polls BSC/Ethereum for new events
    - If event found: Store in DynamoDB with relayer signature
2. **Consensus Validation**:
    - DynamoDB Streams triggers Validator Lambda
    - Validator counts signatures for event
    - If 2-of-3 signatures: Consensus reached → Invoke Executor
3. **Execution**:
    - Executor Lambda receives validated event
    - Executes Mint (Ethereum) or Unlock (BSC)
    - Updates DynamoDB with execution status

💰 Cost Estimation (AWS)

Monthly costs (assuming 1000 transactions/month):

- Lambda invocations: ~$5
- DynamoDB: ~$3
- API Gateway: ~$1
- CloudWatch Logs: ~$2
- Secrets Manager: ~$1

Total: ~$12/month (very low cost due to serverless)

🎯 Demo Walkthrough

1. **Connect MetaMask** to BSC Testnet
2. **Check Balances** on both chains
3. **Bridge BSC → Ethereum**:
    - Enter amount (e.g., 10 tokens)
    - Approve token spending
    - Execute lock transaction
    - Copy Event ID from transaction
4. **Track Status**:
    - Paste Event ID in status tracker
    - Watch relayer signatures accumulate (2-of-3)
    - See execution complete (~1-2 minutes)

5. **Verify**:
    - Check Ethereum balance increased
    - BSC balance decreased
    - Total supply constant

🛠️ Troubleshooting

Common Issues

"Transaction failed"

- Check gas balance on current chain
- Verify correct network selected
- Ensure sufficient token balance

"Consensus not reached"

- Wait 1-2 minutes for all relayers to sign
- Check Lambda function logs
- Verify EventBridge rules are enabled

"Contract not found"

- Verify contract addresses in `.env`
- Ensure contracts deployed to correct network
- Check ABI files are present

Debug Commands

```
# Check Lambda function status
aws lambda get-function --function-name dev-EventPoller

# View DynamoDB table
aws dynamodb scan --table-name dev-BridgeTable --limit 10

# Test API endpoint
curl https://your-api-gateway-url.amazonaws.com/dev/health
```

📝 Environment Variables

Contracts

- `BSC_SEPOLIA_RPC_URL` - BSC RPC endpoint
- `ETHEREUM_SEPOLIA_RPC_URL` - Ethereum RPC endpoint
- `DEPLOYER_PRIVATE_KEY` - Deployer private key

Backend

- `AWS_REGION` - AWS region (default: us-east-1)
- `DYNAMODB_TABLE_NAME` - DynamoDB table name
- Contract addresses (BSC_TOKEN_ADDRESS, etc.)

Frontend

- `VITE_API_GATEWAY_URL` - API Gateway URL
- Contract addresses (VITE_BSC_TOKEN_ADDRESS, etc.)

📚 Additional Resources

- Hardhat Documentation
- AWS SAM Documentation
- DynamoDB Single Table Design
- ethers.js Documentation

🤝 Contributing

This project follows:

- SOLID principles
- Design patterns (Factory, Singleton, Strategy, etc.)
- DRY (Don't Repeat Yourself)
- Clean Code principles

📄 License

MIT License - See LICENSE file for details

Built with: Solidity, OpenZeppelin, Hardhat, AWS SAM, Lambda, DynamoDB, React, Vite, Tailwind CSS, ethers.js

Architecture: Serverless, Multi-Relayer Consensus, Single Table Design