Comprehensive Technical Development Guide: deBridge to Hyperliquid Bridge Implementation

1. deBridge Protocol Overview and Architecture

1.1 Core Architecture Components

deBridge is a cross-chain interoperability protocol Debridge +2 with two primary layers: debridge +5

Protocol Layer (On-Chain):

- **DeBridgeGate**: Main contract at `0x43dE2d77BF8027e25dBD179B491e8d64f38398aA` (consistent across EVM chains) github +4
- **DeBridgeToken**: ERC20 wrapped assets (deAssets) with 1:1 backing GitHub +2
- **SignatureVerifier**: Validates 2/3 validator consensus GitHub +2
- **CallProxy**: Executes cross-chain messages GitHub +2
- **WethGate**: Handles ETH transfers for upgradeable contracts GitHub +2

Infrastructure Layer (Off-Chain):

- Network of independent validators GitHub +3
- Signatures stored on Arweave
- Sub-second finality with HyperBFT consensus DWF Labs +2
- 200,000 orders/second processing capability DWF Labs +3

1.2 Transaction Lifecycle

```
1. User calls deBridgeGate.send() on source chain
2. Submission ID generated (unique hash)
3. Block confirmations (12 blocks for most chains)
4. Validators sign with 2/3 consensus requirement
5. Signatures stored on Arweave
6. Anyone can call deBridgeGate.claim() on destination
7. Execution if signatures valid
```

2. Hyperliquid Network Specifications

2.1 Network Configuration

Mainnet:

```javascript
```

```
{
  chainId: 999,
  rpcUrl: "https://rpc.hyperliquid.xyz/evm",
  nativeToken: "HYPE",
  decimals: 18,
  blockTime: 0.07, // seconds
  gasLimit: {
    small: 2000000,  // Every 1 second
    large: 30000000  // Every 1 minute
  }
}
```

Testnet:

```javascript
{
  chainId: 998,
  rpcUrl: "https://rpc.hyperliquid-testnet.xyz/evm",
  nativeToken: "HYPE",
  decimals: 18
}
```

2.2 Dual Architecture

- **HyperCore**: Order book trading, 200k orders/sec chain +2
- **HyperEVM**: EVM-compatible smart contracts, Cancun hardfork support QuickNode +3
- **Special Address**: `0x2222222222222222222222222222222222222222` for HYPE transfers Hyperliquid Docs gitbook

3. Complete Hardhat Configuration

3.1 Package Dependencies

```json
```

```json
{
  "name": "debridge-hyperliquid-bridge",
  "version": "1.0.0",
  "scripts": {
    "compile": "hardhat compile",
    "test": "hardhat test",
    "deploy": "hardhat run scripts/deploy.js",
    "deploy-multichain": "hardhat run scripts/deployMultichain.js",
    "debridge-emulator": "hardhat debridge-run-emulator --network localhost"
  },
  "devDependencies": {
    "@nomicfoundation/hardhat-toolbox": "^5.0.0",
    "@nomicfoundation/hardhat-ethers": "^3.0.6",
    "@nomicfoundation/hardhat-verify": "^2.0.9",
    "@debridge-finance/hardhat-debridge": "^2.0.0-rc.0",
    "@debridge-finance/desdk": "^1.4.0",
    "@openzeppelin/contracts": "^5.0.2",
    "hardhat": "^2.22.6",
    "hardhat-deploy": "^0.12.4",
    "ethers": "^6.13.2",
    "dotenv": "^16.4.5",
    "typescript": "^5.5.3"
  }
}
```

3.2 Hardhat Configuration

```typescript
```

```
import { HardhatUserConfig } from "hardhat/config";
import "@nomicfoundation/hardhat-toolbox";
import "@debridge-finance/hardhat-debridge";
import "hardhat-deploy";
import "dotenv/config";

const config: HardhatUserConfig = {
  solidity: {
    version: "0.8.28",
    settings: {
      optimizer: {
        enabled: true,
        runs: 200
      },
      evmVersion: "cancun"
    }
  },

  networks: {
    hardhat: {
      chainId: 31337,
      forking: {
        url: process.env.ETHEREUM_RPC_URL || "",
        enabled: false
      }
    },

    mainnet: {
      url: process.env.ETHEREUM_RPC_URL || `https://eth-mainnet.g.alchemy.com/v2/${process.env.ALCHEMY_KEY}`,
      accounts: process.env.PRIVATE_KEY ? [process.env.PRIVATE_KEY] : [],
      chainId: 1
    },

    hyperliquid: {
      url: "https://rpc.hyperliquid.xyz/evm",
      accounts: process.env.PRIVATE_KEY ? [process.env.PRIVATE_KEY] : [],
      chainId: 999,
      gasPrice: "auto"
    },

    hyperliquidTestnet: {
      url: "https://rpc.hyperliquid-testnet.xyz/evm",
      accounts: process.env.PRIVATE_KEY ? [process.env.PRIVATE_KEY] : [],
      chainId: 998
    },
    arbitrum: {
      url: "https://arb1.arbitrum.io/rpc",
      accounts: process.env.PRIVATE_KEY ? [process.env.PRIVATE_KEY] : [],
      chainId: 42161
    },

    base: {
      url: "https://mainnet.base.org",
      accounts: process.env.PRIVATE_KEY ? [process.env.PRIVATE_KEY] : [],
      chainId: 8453
    }
  },

  etherscan: {
    apiKey: {
      mainnet: process.env.ETHERSCAN_API_KEY || "",
      arbitrumOne: process.env.ARBISCAN_API_KEY || "",
      base: process.env.BASESCAN_API_KEY || ""
    }
  }
};

export default config;
```

4. SDK Integration

4.1 deBridge SDK Setup

```
javascript
```

```javascript
import { evm } from "@debridge-finance/desdk";

class DeBridgeClient {
  constructor(provider, signer) {
    this.provider = provider;
    this.signer = signer;
    this.gateAddress = "0x43dE2d77BF8027e25dBD179B491e8d64f38398aA";
  }

  async createBridgeMessage(params) {
    const message = new evm.Message({
      tokenAddress: params.token,
      amount: params.amount,
      chainIdTo: "999", // Hyperliquid
      receiver: params.receiver,
      autoParams: new evm.SendAutoParams({
        executionFee: params.executionFee || "0",
        fallbackAddress: params.receiver,
        flags: new evm.Flags(),
        data: params.callData || "0x"
      })
    });

    return message.getEncodedArgs();
  }

  async trackSubmission(txHash, originContext) {
    const submissions = await evm.Submission.findAll(txHash, originContext);
    const submission = submissions[0];

    const isConfirmed = await submission.hasRequiredBlockConfirmations();
    if (isConfirmed) {
      const claim = await submission.toEVMClaim(destinationContext);
      return await claim.getEncodedArgs();
    }

    return null;
  }
}
```

4.2 DLN API Integration

javascript

```
class DLNClient {
  constructor() {
    this.baseUrl = "https://dln.debridge.finance/v1.0";
  }

  async createOrder(params) {
    const queryParams = new URLSearchParams({
      srcChainId: params.srcChainId,
      srcChainTokenIn: params.tokenIn,
      srcChainTokenInAmount: params.amountIn,
      dstChainId: "999", // Hyperliquid
      dstChainTokenOut: params.tokenOut,
      dstChainTokenOutAmount: params.amountOut || "auto",
      dstChainTokenOutRecipient: params.recipient
    });

    const response = await fetch(`${this.baseUrl}/dln/order/create-tx?${queryParams}`);
    return await response.json();
  }

  async getOrderStatus(orderId) {
    const response = await fetch(`${this.baseUrl}/dln/order/${orderId}`);
    return await response.json();
  }
}
```

5. Smart Contract Implementation

5.1 Core Bridge Contract

```
solidity
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/security/Pausable.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

interface IDeBridgeGate {
    struct SubmissionAutoParamsTo {
        uint256 executionFee;
        uint256 flags;
        bytes fallbackAddress;
        bytes data;
    }

    function send(
        address _tokenAddress,
        uint256 _amount,
        uint256 _chainIdTo,
        bytes memory _receiver,
        bytes memory _permit,
        bool _useAssetFee,
        uint32 _referralCode,
        bytes calldata _autoParams
    ) external payable;

    function globalFixedNativeFee() external view returns (uint256);
    function callProxy() external view returns (address);
}

contract HyperliquidBridge is ReentrancyGuard, Pausable {
    using SafeERC20 for IERC20;

    IDeBridgeGate public immutable deBridgeGate;
    uint256 public constant HYPERLIQUID_CHAIN_ID = 999;

    // Events
    event BridgeInitiated(
        address indexed sender,
        address indexed token,
        uint256 amount,
        address indexed receiver,
        bytes32 submissionId
    );
```

```
event BridgeReceived(
    bytes32 indexed submissionId,
    address indexed recipient,
    address indexed token,
    uint256 amount
);

// Security mappings
mapping(bytes32 => bool) public processedSubmissions;
mapping(address => bool) public supportedTokens;
mapping(uint256 => mapping(bytes32 => bool)) public trustedSenders;

modifier onlyCallProxy() {
    require(msg.sender == deBridgeGate.callProxy(), "Unauthorized");
    _;
}

constructor(address _deBridgeGate) {
    deBridgeGate = IDeBridgeGate(_deBridgeGate);
}

/**
 * @dev Bridge tokens to Hyperliquid
 * @param token Token address (address(0) for native)
 * @param amount Amount to bridge
 * @param receiver Receiver address on Hyperliquid
 * @param referralCode Optional referral code
 */
function bridgeToHyperliquid(
    address token,
    uint256 amount,
    address receiver,
    uint32 referralCode
) external payable nonReentrant whenNotPaused {
    require(supportedTokens[token], "Token not supported");
    require(amount > 0, "Invalid amount");
    require(receiver != address(0), "Invalid receiver");

    // Handle token transfer
    if (token != address(0)) {
        IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
        IERC20(token).forceApprove(address(deBridgeGate), amount);
    }

    // Calculate fees
    uint256 protocolFee = deBridgeGate.globalFixedNativeFee();
    uint256 executionFee = msg.value - protocolFee;
```

```
        if (token == address(0)) {
            // Native token bridging
            require(msg.value >= amount + protocolFee, "Insufficient value");
            executionFee = msg.value - amount - protocolFee;
        } else {
            require(msg.value >= protocolFee, "Insufficient fee");
        }

        // Build auto parameters
        bytes memory autoParams = _buildAutoParams(receiver, executionFee);

        // Execute bridge
        deBridgeGate.send{value: token == address(0) ? msg.value : protocolFee + executionFee}(
            token,
            amount,
            HYPERLIQUID_CHAIN_ID,
            abi.encodePacked(receiver),
            "",
            false,
            referralCode,
            autoParams
        );

        emit BridgeInitiated(msg.sender, token, amount, receiver, keccak256(abi.encode(block.timestamp, msg.sender, am
    }

    /**
     * @dev Receive bridged assets from source chain
     */
    function receiveBridge(
        address recipient,
        address token,
        uint256 amount,
        bytes32 submissionId
    ) external onlyCallProxy nonReentrant {
        require(!processedSubmissions[submissionId], "Already processed");
        require(recipient != address(0), "Invalid recipient");

        processedSubmissions[submissionId] = true;

        if (token != address(0)) {
            IERC20(token).safeTransfer(recipient, amount);
        } else {
            (bool success, ) = payable(recipient).call{value: amount}("");
            require(success, "Transfer failed");
        }
        emit BridgeReceived(submissionId, recipient, token, amount);
    }

    function _buildAutoParams(address receiver, uint256 executionFee)
        internal
        pure
        returns (bytes memory)
    {
        uint256 flags = 1; // REVERT_IF_EXTERNAL_FAIL

        return abi.encode(
            IDeBridgeGate.SubmissionAutoParamsTo({
                executionFee: executionFee,
                flags: flags,
                fallbackAddress: abi.encodePacked(receiver),
                data: ""
            })
        );
    }

    // Admin functions
    function setSupportedToken(address token, bool supported) external {
        supportedTokens[token] = supported;
    }

    function pause() external {
        _pause();
    }

    function unpause() external {
        _unpause();
    }
}
```

5.2 Gas-Optimized Implementation

solidity

```
contract OptimizedHyperliquidBridge {
    using SafeERC20 for IERC20;

    // Packed struct for gas efficiency
    struct BridgeData {
        address token;      // 20 bytes
        uint96 amount;      // 12 bytes - same slot
        address receiver;   // 20 bytes
        uint32 chainId;     // 4 bytes
        uint32 nonce;       // 4 bytes
        uint16 referral;    // 2 bytes - same slot
    }

    // Assembly optimization for reading calldata
    function unpackBridgeData(bytes calldata data)
        internal
        pure
        returns (BridgeData memory)
    {
        BridgeData memory bd;
        assembly {
            let ptr := add(data.offset, 0x20)
            bd := mload(0x40)

            mstore(bd, calldataload(ptr))           // token
            mstore(add(bd, 0x20), calldataload(add(ptr, 0x20)))  // amount
            mstore(add(bd, 0x40), calldataload(add(ptr, 0x40)))  // receiver
            mstore(add(bd, 0x60), calldataload(add(ptr, 0x60)))  // chainId + nonce + referral
        }
        return bd;
    }

    // Batch bridging for multiple tokens
    function batchBridge(BridgeData[] calldata bridges)
        external
        payable
    {
        uint256 totalFee = deBridgeGate.globalFixedNativeFee() * bridges.length;
        require(msg.value >= totalFee, "Insufficient fee");

        for (uint256 i; i < bridges.length;) {
            _executeBridge(bridges[i]);

            unchecked { ++i; }
        }
    }
}
```

6. Deployment Scripts

6.1 Multi-Chain Deployment

```
typescript
```

```
// scripts/deploy-multichain.ts
import { ethers, network } from "hardhat";
import fs from "fs";

const DEBRIDGE_GATES = {
  mainnet: "0x43dE2d77BF8027e25dBD179B491e8d64f38398aA",
  arbitrum: "0x43dE2d77BF8027e25dBD179B491e8d64f38398aA",
  base: "0x43dE2d77BF8027e25dBD179B491e8d64f38398aA",
  polygon: "0x43dE2d77BF8027e25dBD179B491e8d64f38398aA",
  optimism: "0x43dE2d77BF8027e25dBD179B491e8d64f38398aA"
};

async function deployBridge() {
  console.log(`Deploying to ${network.name}...`);

  const [deployer] = await ethers.getSigners();
  console.log("Deploying with account:", deployer.address);

  const deBridgeGate = DEBRIDGE_GATES[network.name];
  if (!deBridgeGate) {
    throw new Error(`No deBridge gate for ${network.name}`);
  }

  // Deploy bridge contract
  const Bridge = await ethers.getContractFactory("HyperliquidBridge");
  const bridge = await Bridge.deploy(deBridgeGate);
  await bridge.waitForDeployment();

  const bridgeAddress = await bridge.getAddress();
  console.log("Bridge deployed to:", bridgeAddress);

  // Configure supported tokens
  const USDC = {
    mainnet: "0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48",
    arbitrum: "0xaf88d065e77c8cC2239327C5EDb3A432268e5831",
    base: "0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913"
  };

  if (USDC[network.name]) {
    await bridge.setSupportedToken(USDC[network.name], true);
    console.log("USDC support enabled");
  }

  // Save deployment info
  const deploymentInfo = {
    network: network.name,
    chainId: network.config.chainId,
    bridge: bridgeAddress,
    deBridgeGate,
    timestamp: new Date().toISOString()
  };

  const deploymentPath = `./deployments/${network.name}.json`;
  fs.writeFileSync(deploymentPath, JSON.stringify(deploymentInfo, null, 2));

  return bridgeAddress;
}

async function main() {
  const networks = ["mainnet", "arbitrum", "base"];
  const deployments = {};

  for (const net of networks) {
    const address = await deployBridge();
    deployments[net] = address;
  }

  console.log("\nAll deployments:", deployments);
}

main()
  .then(() => process.exit(0))
  .catch(error => {
    console.error(error);
    process.exit(1);
  });
```

7. Testing Strategies

7.1 Unit Tests

javascript

```
const { expect } = require("chai");
const { ethers } = require("hardhat");
const { loadFixture } = require("@nomicfoundation/hardhat-network-helpers");

describe("HyperliquidBridge", function() {
  async function deployFixture() {
    const [owner, user, receiver] = await ethers.getSigners();

    // Deploy mock deBridge gate
    const MockGate = await ethers.getContractFactory("MockDeBridgeGate");
    const gate = await MockGate.deploy();

    // Deploy bridge
    const Bridge = await ethers.getContractFactory("HyperliquidBridge");
    const bridge = await Bridge.deploy(gate.address);

    // Deploy test token
    const Token = await ethers.getContractFactory("MockERC20");
    const token = await Token.deploy("Test", "TEST");

    // Setup
    await bridge.setSupportedToken(token.address, true);
    await token.mint(user.address, ethers.parseEther("1000"));

    return { bridge, gate, token, owner, user, receiver };
  }

  describe("Bridge Operations", function() {
    it("Should bridge tokens successfully", async function() {
      const { bridge, token, user, receiver } = await loadFixture(deployFixture);

      const amount = ethers.parseEther("100");
      const fee = ethers.parseEther("0.01");

      await token.connect(user).approve(bridge.address, amount);

      await expect(
        bridge.connect(user).bridgeToHyperliquid(
          token.address,
          amount,
          receiver.address,
          0,
          { value: fee }
        )
      ).to.emit(bridge, "BridgeInitiated")
        .withArgs(user.address, token.address, amount, receiver.address);
      expect(await token.balanceOf(bridge.address)).to.equal(amount);
    });

    it("Should handle native token bridging", async function() {
      const { bridge, user, receiver } = await loadFixture(deployFixture);

      await bridge.setSupportedToken(ethers.ZeroAddress, true);
      const amount = ethers.parseEther("1");
      const fee = ethers.parseEther("0.01");

      await expect(
        bridge.connect(user).bridgeToHyperliquid(
          ethers.ZeroAddress,
          amount,
          receiver.address,
          0,
          { value: amount + fee }
        )
      ).to.emit(bridge, "BridgeInitiated");
    });
  });
});
```

7.2 Integration Tests

javascript

```
describe("Cross-Chain Integration", function() {
  it("Should complete full bridge cycle", async function() {
    const { sourceBridge, destBridge, token } = await loadFixture(deployIntegrationFixture);

    // Step 1: Initiate bridge on source chain
    const amount = ethers.parseEther("50");
    await token.approve(sourceBridge.address, amount);
    const tx = await sourceBridge.bridgeToHyperliquid(
      token.address,
      amount,
      destBridge.address,
      0
    );

    // Step 2: Get submission data
    const receipt = await tx.wait();
    const event = receipt.logs.find(log => log.fragment.name === "BridgeInitiated");

    // Step 3: Simulate validator signatures (mock)
    const signatures = await mockValidatorSignatures(event.args);

    // Step 4: Execute on destination
    await destBridge.receiveBridge(
      user.address,
      token.address,
      amount,
      event.args.submissionId
    );

    // Verify final state
    expect(await token.balanceOf(user.address)).to.equal(amount);
  });
});
```

8. Error Handling

8.1 Common Error Patterns

```
solidity
```

```
library BridgeErrors {
    error InvalidToken(address token);
    error InsufficientAmount(uint256 provided, uint256 required);
    error UnsupportedChain(uint256 chainId);
    error ExecutionFailed(bytes32 submissionId);
    error UnauthorizedSender(address sender);
    error ExpiredTransaction(uint256 deadline);
}

contract ErrorHandlingBridge {
    function bridgeWithValidation(
        address token,
        uint256 amount,
        address receiver
    ) external payable {
        if (!supportedTokens[token])
            revert BridgeErrors.InvalidToken(token);

        if (amount == 0)
            revert BridgeErrors.InsufficientAmount(0, 1);

        if (receiver == address(0))
            revert BridgeErrors.UnauthorizedSender(receiver);

        // Bridge logic
    }
}
```

8.2 Error Recovery

javascript

```
class BridgeErrorHandler {
  async handleBridgeError(error, context) {
    const errorPatterns = {
      "Insufficient fee": () => this.handleInsufficientFee(context),
      "Token not supported": () => this.handleUnsupportedToken(context),
      "Already processed": () => this.handleDuplicateSubmission(context),
      "Rate limit exceeded": () => this.handleRateLimit(context)
    };

    for (const [pattern, handler] of Object.entries(errorPatterns)) {
      if (error.message.includes(pattern)) {
        return await handler();
      }
    }

    throw error; // Re-throw unknown errors
  }

  async handleInsufficientFee(context) {
    const requiredFee = await this.calculateRequiredFee(context);
    return {
      retry: true,
      suggestedFee: requiredFee,
      message: `Increase fee to ${ethers.formatEther(requiredFee)} ETH`
    };
  }
}
```

9. Security Best Practices

9.1 Multi-Signature Security

solidity

```
contract SecureBridge {
    uint256 public constant SIGNATURE_THRESHOLD = 2;
    mapping(address => bool) public validators;

    modifier requireMultiSig(bytes32 txHash, bytes[] memory signatures) {
        uint256 validSigs = 0;

        for (uint256 i = 0; i < signatures.length; i++) {
            address signer = ECDSA.recover(txHash, signatures[i]);
            if (validators[signer]) validSigs++;
        }

        require(validSigs >= SIGNATURE_THRESHOLD, "Insufficient signatures");
        _;
    }
}
```

9.2 Rate Limiting

solidity

```
contract RateLimitedBridge {
    mapping(address => uint256) public dailyLimit;
    mapping(address => uint256) public dailyUsed;
    mapping(address => uint256) public lastReset;

    modifier rateLimited(uint256 amount) {
        if (block.timestamp > lastReset[msg.sender] + 1 days) {
            dailyUsed[msg.sender] = 0;
            lastReset[msg.sender] = block.timestamp;
        }

        require(
            dailyUsed[msg.sender] + amount <= dailyLimit[msg.sender],
            "Daily limit exceeded"
        );

        dailyUsed[msg.sender] += amount;
        _;
    }
}
```

10. Gas Optimization

10.1 Storage Optimization

```solidity
// Before: 3 storage slots
struct UnoptimizedData {
    address user;      // 32 bytes
    uint256 amount;    // 32 bytes
    uint256 timestamp; // 32 bytes
}

// After: 2 storage slots
struct OptimizedData {
    address user;      // 20 bytes
    uint96 amount;     // 12 bytes - same slot
    uint256 timestamp; // 32 bytes
}
```

10.2 Calldata Optimization

```solidity
// Use calldata for read-only arrays
function processBridgeData(BridgeData[] calldata data) external {
    // Process without copying to memory
}

// Pack multiple values
function packData(address token, uint96 amount, uint32 chainId)
    pure
    returns (bytes32)
{
    return bytes32(abi.encodePacked(token, amount, chainId));
}
```

11. Transaction Monitoring

11.1 Event Monitoring

```javascript
```

```
class BridgeMonitor {
  constructor(bridgeContract, provider) {
    this.bridge = bridgeContract;
    this.provider = provider;
  }

  async startMonitoring() {
    // Monitor bridge initiations
    this.bridge.on("BridgeInitiated", async (sender, token, amount, receiver, submissionId) => {
      console.log(`Bridge initiated: ${submissionId}`);

      // Track transaction
      await this.trackTransaction({
        submissionId,
        sender,
        token,
        amount,
        receiver,
        timestamp: Date.now()
      });
    });

    // Monitor completions
    this.bridge.on("BridgeReceived", async (submissionId, recipient, token, amount) => {
      console.log(`Bridge completed: ${submissionId}`);

      await this.updateTransactionStatus(submissionId, "completed");
    });
  }

  async trackTransaction(data) {
    // Store in database or monitoring system
  }
}
```

11.2 Health Monitoring

javascript

```
class BridgeHealthMonitor {
  async checkBridgeHealth() {
    const checks = {
      contractDeployed: await this.checkContractDeployment(),
      validatorStatus: await this.checkValidatorStatus(),
      gasPrice: await this.checkGasPrice(),
      networkLatency: await this.checkNetworkLatency()
    };

    return {
      healthy: Object.values(checks).every(c => c.status === "ok"),
      checks
    };
  }
}
```

12. API Integration

12.1 REST API Endpoints

javascript

```
// Express API for bridge operations
const express = require('express');
const app = express();

app.post('/api/bridge/initiate', async (req, res) => {
  try {
    const { token, amount, receiver } = req.body;

    // Validate input
    if (!isValidAddress(token) || !isValidAddress(receiver)) {
      return res.status(400).json({ error: "Invalid address" });
    }

    // Execute bridge
    const tx = await bridge.bridgeToHyperliquid(
      token,
      amount,
      receiver,
      0
    );

    res.json({
      success: true,
      txHash: tx.hash,
      submissionId: calculateSubmissionId(tx)
    });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.get('/api/bridge/status/:submissionId', async (req, res) => {
  const status = await getTransactionStatus(req.params.submissionId);
  res.json(status);
});
```

13. Common Issues and Solutions

13.1 Troubleshooting Guide

Issue	Cause	Solution
"Insufficient fee"	Gas price spike	Increase fee by 20%
"Token not supported"	Token not whitelisted	Add token via setSupportedToken()
"Already processed"	Duplicate submission	Check submission ID uniqueness
"Rate limit exceeded"	Too many requests	Implement exponential backoff

Issue	Cause	Solution
"Invalid signature"	Wrong signer	Verify validator addresses
"Network unavailable"	RPC issues	Use fallback RPC providers

13.2 Debug Utilities

javascript

```
// Debug transaction
async function debugBridgeTransaction(txHash) {
  const tx = await provider.getTransaction(txHash);
  const receipt = await provider.getTransactionReceipt(txHash);

  console.log("Transaction Details:");
  console.log("- Status:", receipt.status ? "Success" : "Failed");
  console.log("- Gas Used:", receipt.gasUsed.toString());
  console.log("- Block:", receipt.blockNumber);

  // Decode logs
  for (const log of receipt.logs) {
    try {
      const parsed = bridge.interface.parseLog(log);
      console.log(`Event: ${parsed.name}`, parsed.args);
    } catch (e) {
      // Unknown event
    }
  }

  // Get revert reason if failed
  if (!receipt.status) {
    const reason = await getRevertReason(txHash);
    console.log("Revert Reason:", reason);
  }
}
```

Appendix A: Quick Reference

Chain IDs

- Ethereum: 1
- Arbitrum: 42161
- Base: 8453
- Polygon: 137
- Hyperliquid: 999

- Hyperliquid Testnet: 998

deBridge Gate Addresses

- All EVM chains: `0x43dE2d77BF8027e25dBD179B491e8d64f38398aA`

Gas Limits

- Simple bridge: 200,000
- Complex bridge with callback: 500,000
- Batch operations: 150,000 per item

Fee Structure

- Protocol fee: 0.001 ETH (Ethereum)
- Execution fee: Variable based on destination gas
- Hyperliquid withdrawal: 1 USDC

Appendix B: Security Checklist

□ Multi-signature implementation
□ Rate limiting enabled
□ Pause mechanism deployed
□ Input validation complete
□ Reentrancy guards active
□ Emergency withdrawal tested
□ Monitoring system online
□ Incident response plan ready
□ Audit completed
□ Bug bounty program active

This comprehensive guide provides all necessary technical details for implementing a secure, efficient deBridge to Hyperliquid bridge system using modern development practices and tools.