Context Document for LLM: Coinbase Developer Platform (CDP) Wallet API v2 & Node.js SDK (`@coinbase/cdp-sdk`)

1. Overview of Wallet API v2 & `@coinbase/cdp-sdk`

- **Purpose:** The Wallet API v2 enables developers to programmatically create, manage, and use crypto accounts (EVM & Solana). CDP secures private keys within a Trusted Execution Environment (TEE) (e.g., AWS Nitro Enclaves), handling complex infrastructure. Developers interact via the `@coinbase/cdp-sdk` (for TypeScript/Node.js, Python also available) or REST endpoints.
- **Key Benefits of v2 over v1:**
  - **Security Management:** Simplified; keys secured in TEE, not developer-managed.
  - **Authentication:** Single `Wallet Secret` for all accounts (EVM & Solana), plus a standard `Secret API Key` for general API auth.
  - **Network Support:** EVM and Solana.
  - **EVM Account Scope:** Accounts are compatible across multiple EVM chains.
  - **Advanced Features:** Native EIP-4337 Smart Account support enabling transaction batching, gas sponsorship (Paymaster), and spend permissions.
  - **`viem` Compatibility:** CDP EVM accounts can be wrapped into `viem` Custom Accounts. `viem` local accounts can own CDP Smart Accounts.
- **Target SDK (for this context):** `@coinbase/cdp-sdk` for Node.js/TypeScript.
  - Installation: `npm install @coinbase/cdp-sdk dotenv viem` (`viem` is used for utilities and transaction confirmation examples).
  - Repository: `github.com/coinbase/cdp-sdk`

2. Prerequisites & Project Setup (TypeScript/Node.js)

- **Node.js:** Version 22.x+ recommended.
- **CDP Account:** Active Coinbase Developer Platform account.
- **API Credentials (from CDP Portal):**
  1. **Secret API Key:** Consists of `CDP_API_KEY_ID` (Key Name/ID) and `CDP_API_KEY_SECRET` (private key string). Used for general API authentication.
  2. **Wallet Secret:** `CDP_WALLET_SECRET`. A distinct secret for authorizing sensitive Wallet API v2 operations (account creation, signing) via the TEE.
- **Project Initialization Steps:**
  1. Create project directory: `mkdir my-cdp-agent && cd my-cdp-agent`
  2. Initialize npm: `npm init -y`
  3. Set package type for ES Modules: `npm pkg set type="module"`

4. Install TypeScript & types: `npm install typescript @types/node --save-dev`

Create `tsconfig.json`: `npx tsc --init`. Configure for modern Node.js (ES2022, NodeNext module/resolution, rootDir ./src, outDir ./dist).

JSON

```
// tsconfig.json (example)
{
  "compilerOptions": {
    "target": "ES2022", "module": "NodeNext", "moduleResolution": "NodeNext",
    "esModuleInterop": true, "forceConsistentCasingInFileNames": true,
    "strict": true, "skipLibCheck": true,
    "outDir": "./dist", "rootDir": "./src"
  },
  "include": ["src/**/*"], "exclude": ["node_modules"]
}
```

5. Create source files: `mkdir src && touch src/main.ts` (or `agent.ts`, `server.ts` etc.)

Create `.env` file for credentials:

Code snippet

```
CDP_API_KEY_ID=your-api-key-id
CDP_API_KEY_SECRET=your-api-key-secret
CDP_WALLET_SECRET=your-wallet-secret
# Optional: PAYMASTER_RPC_URL=your_base_sepolia_paymaster_url_here
```

6. Install core dependencies: `npm install @coinbase/cdp-sdk dotenv viem`

3. SDK Client Initialization (`CdpClient`)

- The `CdpClient` is the main entry point for interacting with the SDK.

It automatically loads `CDP_API_KEY_ID`, `CDP_API_KEY_SECRET`, and `CDP_WALLET_SECRET` from `process.env` if the parameter-less constructor is used after `dotenv.config()`.

TypeScript

```
// src/main.ts (example)
import { CdpClient } from "@coinbase/cdp-sdk";
import dotenv from "dotenv";

dotenv.config(); // Load .env variables
```

```
const cdp = new CdpClient(); // Initializes by reading from process.env

// Optional verification
if (!process.env.CDP_API_KEY_ID) { /* handle error */ }

console.log("CDP Client Initialized.");
```

-
- Alternatively, credentials can be passed directly: `new CdpClient({ apiKeyId, apiKeySecret, walletSecret });`

4. Managing Accounts (EOAs & Smart Accounts)

- **Account Naming:** Accounts can be assigned names (alphanumeric, hyphens, 2-36 chars, unique per project per account type).

`cdp.evm.createAccount({ name?: string })`: Creates a new EVM EOA.

```
TypeScript
// const ownerEoa = await cdp.evm.createAccount({ name: "MyAgentOwnerEOA" });
// console.log(`Created EVM EOA: ${ownerEoa.address}, UUID: ${ownerEoa.accountUuid}`);
```

-

`cdp.evm.getOrCreateAccount({ name: string })`: Retrieves an existing EVM EOA by name or creates it if it doesn't exist.

```
TypeScript
// const existingOrNewEoa = await cdp.evm.getOrCreateAccount({ name: "MyPersistentEOA" });
```

-

`cdp.evm.createSmartAccount({ owner: EvmAccount, networkId?: string })`: Creates an EVM Smart Account (EIP-4337). Requires an `EvmAccount` object as the owner. Smart Accounts are initially supported on Base Sepolia and Base Mainnet. The smart contract is deployed on the first UserOperation.

```
TypeScript
// Assuming 'ownerEoa' is an EvmAccount object from createAccount or getOrCreateAccount
// const smartAccount = await cdp.evm.createSmartAccount({ owner: ownerEoa });
// console.log(`Created Smart Account: ${smartAccount.address} owned by ${ownerEoa.address}`);
```

-
- **`cdp.solana.createAccount({ name?: string })` / `getOrCreateAccount()`:** Similar methods for Solana accounts.

- **`cdp.evm.listAccounts({ pageToken?: string })`:** Lists EVM accounts with pagination.
- **`cdp.evm.importAccount({ privateKey: Hex, name?: string })`:** Imports an EVM account from a raw hex private key. End-to-end encrypted to the TEE.

5. Funding Accounts (Testnets)

- CDP provides a Faucet API for testnet tokens. Rate limits apply.

For EOAs (`EvmAccount` instance): The `cdp.evm.requestFaucet()` method is used.

```
 TypeScript
// Assuming 'eoaAccount' is an EvmAccount object
// const faucetResponse = await cdp.evm.requestFaucet({
//   address: eoaAccount.address,
//   network: "base-sepolia", // e.g., Base Sepolia
//   token: "eth"             // "eth" is supported for EVM
// });
// console.log(`ETH Faucet request: ${faucetResponse.transactionHash}`);
// // Wait for confirmation using a viem publicClient:
// // const viemPublicClient = createPublicClient({ chain: baseSepolia, transport: http() });
// // await viemPublicClient.waitForTransactionReceipt({ hash: faucetResponse.transactionHash
});
```

-

For Smart Accounts (`EvmSmartAccount` instance): The Smart Account object itself has a `requestFaucet` method.

```
 TypeScript
// Assuming 'mySmartAccount' is an EvmSmartAccount object
// const { transactionHash } = await mySmartAccount.requestFaucet({
//   network: "base-sepolia",
//   token: "eth",
// });
// console.log(`Smart Account ETH Faucet request: ${transactionHash}`);
// // Wait for confirmation using viemPublicClient
// // await viemPublicClient.waitForTransactionReceipt({ hash: transactionHash });
```

-
- **Solana Faucet:** `cdp.solana.requestFaucet({ address, token: "sol" })`.

6. Sending Transactions & UserOperations (EVM)

- **From EOA (`cdp.evm.sendTransaction`):**
    - Handles gas estimation, nonce management, signing (in TEE), and broadcasting.

    - `address` parameter is the *sender's EOA address* (must be managed by this CDP project).
    - `transaction` object takes `to`, `value` (as BigInt via `parseEther`), `data` (optional `0x` or encoded call).
    - `network` parameter specifies the target chain (e.g., "base-sepolia").

<!-- end list -->

TypeScript
```
// Assuming 'senderEoaAddress' is the address of an EOA created via this CDP client
// const ethTxResult = await cdp.evm.sendTransaction({
//   address: senderEoaAddress,
//   network: "base-sepolia",
//   transaction: {
//     to: "0xRecipientAddress",
//     value: parseEther("0.0001"), // from viem
//   },
// });
// console.log(`ETH Tx sent: ${ethTxResult.transactionHash}`);
// // Wait for confirmation using viemPublicClient.waitForTransactionReceipt({ hash: ethTxResult.transactionHash });

// For ERC20 transfer from EOA:
// const erc20Amount = parseUnits("10", 6); // 10 USDC (6 decimals)
// const usdcContractAddress = "0xUSDC_ADDRESS_ON_BASE_SEPOLIA";
// const erc20TransferData = encodeFunctionData({ ... erc20 transfer ABI, args ... });
// const erc20TxResult = await cdp.evm.sendTransaction({
//   address: senderEoaAddress,
//   network: "base-sepolia",
//   transaction: {
//     to: usdcContractAddress, // Token contract is 'to'
//     value: 0n,               // No ETH value for standard ERC20 transfer
//     data: erc20TransferData,
//   },
// });
// console.log(`ERC20 Tx sent: ${erc20TxResult.transactionHash}`);
```

-
- **From Smart Account (`smartAccount.sendUserOperation` or `cdp.evm.sendUserOperation`):**

    - Executes actions via EIP-4337 UserOperations. On Base Sepolia, these are typically gasless by default (subsidized by CDP).

- The `calls` field is an array, enabling **atomic batching** of multiple operations (e.g., multiple ETH transfers, ERC20 transfers, or contract calls).
- `paymasterUrl` (or `paymasterOptions: { url }`) can be used to specify a different Paymaster.
- **Method Invocation:** The docs show examples using *both* `await smartAccount.sendUserOperation(...)` and `await cdp.evm.sendUserOperation(...)`. The methods on the `smartAccount` object instance (like `smartAccount.sendUserOperation` and `smartAccount.requestFaucet`) are often cleaner if you have the object.

<!-- end list -->

TypeScript

```
// Assuming 'mySmartAccount' is an EvmSmartAccount object
// Example 1: Simple ETH transfer UserOperation using smartAccount.sendUserOperation
// const opResult1 = await mySmartAccount.sendUserOperation({
//   network: "base-sepolia", // Network for the UserOp
//   calls: [{
//     to: "0xRecipientAddress",
//     value: parseEther("0.00001"),
//     data: "0x",
//   }],
//   // paymasterOptions: { url: process.env.PAYMASTER_RPC_URL } // For custom/CDP Paymaster
// });
// console.log(`UserOp submitted via smartAccount method. Status: ${opResult1.status}, Hash: ${opResult1.userOpHash}`);

// Example 2: Batch ETH transfers using cdp.evm.sendUserOperation
// const destinationAddresses = ["0xAddr1", "0xAddr2"];
// const callsForBatch = destinationAddresses.map(dest => ({
//   to: dest as `0x${string}`,
//   value: parseEther("0.000001"),
//   data: "0x" as Hex,
// }));
// const opResult2 = await cdp.evm.sendUserOperation({
//   smartAccount: mySmartAccount, // Pass the smartAccount object
//   network: "base-sepolia",
//   calls: callsForBatch,
// });
// console.log(`Batch UserOp submitted via cdp.evm method. Status: ${opResult2.status}, Hash: ${opResult2.userOpHash}`);
```

-

7. Waiting for UserOperation Confirmation

- After `sendUserOperation`, you get a `UserOperationResult` containing `userOpHash` and an initial `status` (e.g., "broadcast").

- To wait for onchain confirmation, use `smartAccount.waitForUserOperation({ userOpHash })` OR `cdp.evm.waitForUserOperation({ smartAccountAddress, userOpHash, network? })`. The method on the `smartAccount` instance is often more convenient.

This returns a `UserOperation` object (or similar, type might be `TransactionReceipt` from SDK) which includes the final `status` (e.g., "complete") and the actual `transactionHash` that included the UserOperation.

TypeScript
```
// Assuming 'mySmartAccount' and 'opResult' (from sendUserOperation)
// if (opResult.userOpHash) {
//   console.log(`Waiting for UserOperation ${opResult.userOpHash} to be confirmed...`);
//   const confirmedUserOp = await mySmartAccount.waitForUserOperation({
//     userOpHash: opResult.userOpHash,
//     // network: "base-sepolia" // often implicit if called on smartAccount instance
//   });
//   if (confirmedUserOp.status === "complete") {
//     console.log(`UserOp confirmed! Onchain Tx Hash: ${confirmedUserOp.transactionHash}`);
//   } else {
//     console.log(`UserOp processing failed or status: ${confirmedUserOp.status}`);
//   }
// }
```

-

8. `account.transfer()` Convenience Method

- Both `EvmAccount` and `EvmSmartAccount` objects have a `transfer()` method for simplified ETH or ERC-20 token transfers.

- It handles encoding and calling `sendTransaction` (for EOAs) or `sendUserOperation` (for Smart Accounts) internally.

- `token` parameter can be `"eth"`, `"usdc"`, or a specific token contract address.

- `amount` can be a human-readable string (which `parseUnits` would handle internally based on token type/decimals known to SDK) or a BigInt in atomic units.

For Smart Account transfers, this method returns `{ userOpHash }`. For EOA transfers, it returns `{ transactionHash }`.

TypeScript
```
// Assuming 'senderEoa' is an EvmAccount, 'receiverAddress' is a string/Address
// const { transactionHash: eoaTxHash } = await senderEoa.transfer({
//   to: receiverAddress,
//   amount: parseEther("0.00001"), // OR "0.00001" if SDK handles eth parsing
//   token: "eth",
//   network: "base-sepolia"
// });
// await viemPublicClient.waitForTransactionReceipt({ hash: eoaTxHash });

// Assuming 'senderSmartAccount' is an EvmSmartAccount
// const { userOpHash } = await senderSmartAccount.transfer({
//   to: receiverAddress,
//   amount: parseUnits("0.01", 6), // For USDC (0.01 USDC, 6 decimals) OR "0.01"
//   token: "usdc", // or USDC contract address
//   network: "base-sepolia"
// });
// const receipt = await senderSmartAccount.waitForUserOperation({ userOpHash });
```

-

9. Message Signing (`cdp.evm.signTypedData`)

- Supports EIP-712 for typed structured data signing.

Requires `address` (of the CDP-managed EOA/SA to sign with), `domain`, `types`, `primaryType`, `message`.

TypeScript
```
// Assuming 'signingAccount' is an EvmAccount or EvmSmartAccount object
// const signature = await cdp.evm.signTypedData({
//   address: signingAccount.address,
//   domain: { name: "MyDApp", chainId: 84532, verifyingContract: "0x..." },
//   types: { Person: [{ name: "name", type: "string" }, { name: "wallet", type: "address" }] },
```

```
//   primaryType: "Person",
//   message: { name: "Alice", wallet: "0x..." },
// });
// console.log("EIP-712 Signature:", signature);
```

-

10. `viem` Compatibility

CDP Account as `viem` Custom Account: Use `toAccount(cdpEvmAccount)` from `viem/accounts` to wrap a CDP `EvmAccount` object. This wrapped account can then be used with `viem`'s `createWalletClient`. The `walletClient` can then be used to send transactions, which will be signed via CDP's TEE.

```
 TypeScript
// import { toAccount } from "viem/accounts";
// import { createWalletClient, http } from "viem";
// const cdpEvmAcc = await cdp.evm.createAccount();
// const viemWalletClient = createWalletClient({
//   account: toAccount(cdpEvmAcc), // Wrap CDP account
//   chain: baseSepolia,
//   transport: http() // Transport for broadcasting, signing is via CDP
// });
// const hash = await viemWalletClient.sendTransaction({ to: "0x...", value: parseEther("0.001")
});
```

-
- **`viem` Local Account as Smart Account Owner:** A `viem` `LocalAccount` (e.g., from `privateKeyToAccount`) can be passed as the `owner` when creating a `cdp.evm.createSmartAccount({ owner: viemLocalAccount })`.

11. Policies

- CDP Wallets support Policies to govern account/project behavior (transaction filtering, allowlists, limits).
- Defined by `scope` (project/account), `rules` (action, operation, criteria).
- Project policies evaluated first, then account policies.
- Supported operations for policies: `signEvmTransaction`, `sendEvmTransaction`, `signSolTransaction`.
- Can be created via CDP Portal UI or SDK: `cdp.policies.createPolicy({...})`.
- Apply to account: `cdp.evm.updateAccount({ address, update: { accountPolicy: policyId } })`.

Key Security & Operational Points from Docs:

- **TEE:** Private keys are generated and signing happens within AWS Nitro Enclaves TEE; keys never exposed.
- **Wallet Secret:** Used to authenticate sensitive requests to the TEE. Rotatable.
- **2FA:** Recommended for account security.
- **Node.js Version Error (`ERR_REQUIRE_ESM`):** Use Node v20.19.0+ as CDP SDK v6+ depends on `jose` v6 which is ESM-only.
- **Error Reporting:** Can be disabled with `DISABLE_CDP_ERROR_REPORTING=true` env var.

This context document should provide the LLM with a solid, up-to-date understanding of how to use the `@coinbase/cdp-sdk` for the functionalities planned in your "Backend Wallet Service" video.

Below are the examples a backend service using CDP SDK:

```
import { CdpClient } from "@coinbase/cdp-sdk";

let cdpInstance: CdpClient;

export function initializeCdpClient(): CdpClient {

  if (cdpInstance) return cdpInstance;

  const { CDP_API_KEY_ID, CDP_API_KEY_SECRET, CDP_WALLET_SECRET } =
process.env;

  if (!CDP_API_KEY_ID || !CDP_API_KEY_SECRET || !CDP_WALLET_SECRET) {

    console.error("CRITICAL: CDP credentials missing in .env");

    throw new Error("Server config error: Missing CDP credentials.");

  }

  cdpInstance = new CdpClient({ apiKeyId: CDP_API_KEY_ID, apiKeySecret:
CDP_API_KEY_SECRET, walletSecret: CDP_WALLET_SECRET });

  console.log("CDP Client initialized for Wallet Service.");

  return cdpInstance;

}

export function getCdpClient(): CdpClient {
```

```
  if (!cdpInstance) throw new Error("CDP Client not initialized."); // Should be caught by startup call
  return cdpInstance;
}

import express from 'express';
import dotenv from 'dotenv';
import { initializeCdpClient, getCdpClient } from './cdpClient.js';
import { parseEther, isAddress, createPublicClient, http, Hex } from 'viem'; // Removed encodeFunctionData for now
import { baseSepolia } from 'viem/chains';
import type { EvmAccount as CdpEvmAccount } from '@coinbase/cdp-sdk';

dotenv.config();
initializeCdpClient(); // Initialize CDP client when server starts

const app = express();
app.use(express.json()); // Middleware to parse JSON request bodies
const PORT = process.env.PORT || 3002;

// For demo: NOT FOR PRODUCTION - Use a proper database!
interface ManagedWallet {
  serviceId: string;         // Unique ID generated by this service
  cdpAccountUuid: string;    // UUID from CDP for the account
  address: `0x${string}`;    // Blockchain address of the EOA
  name?: string;             // Name given to the wallet in CDP (e.g., UserWallet-username-network)
```

```
  network: string;           // Network context (e.g., "base-sepolia")
  createdAt: Date;
  userIdentifier?: string;   // Optional: The username or dApp user ID this wallet is for
}
const managedWalletsStore: ManagedWallet[] = [];
let internalWalletIdCounter = 0; // Simple ID generator for serviceId

// viem public client for reading blockchain data (like waiting for receipts)
const viemPublicClient = createPublicClient({ chain: baseSepolia, transport: http() });

app.get('/', (req, res) => res.send('CDP Wallet Service Running'));

// --- API Endpoints will be added below ---

app.listen(PORT, () => console.log(`CDP Wallet Service listening on port ${PORT}`));

// In src/server.ts, where "// --- API Endpoints will be added below ---" is:
app.post('/wallets', async (req, res) => {
  console.log('[API /wallets] Request to create general service wallet.');
  const { name, assignedTo, network = "base-sepolia" } = req.body;

  if (!assignedTo || typeof assignedTo !== 'string' || assignedTo.trim() === '') {
    return res.status(400).json({ error: "'assignedTo' identifier (string) is required." });
  }
  if (typeof network !== 'string' || !network) {
```

```
    return res.status(400).json({ error: "Valid 'network' (string) is required."});
  }

  try {
    const cdp = getCdpClient();
    const walletNameForCDP = name || `ServiceWallet-${assignedTo}-${Date.now()}`;
    console.log(`[API /wallets] Calling cdp.evm.createAccount() with name for CDP: ${walletNameForCDP}`);

    const newCdpAccount = await cdp.evm.createAccount({ name: walletNameForCDP });

    internalWalletIdCounter++;
    const newManagedWallet: ManagedWallet = {
      serviceId: `srv-wallet-${internalWalletIdCounter}`,
      cdpAccountUuid: newCdpAccount.accountUuid,
      address: newCdpAccount.address,
      name: newCdpAccount.name, // This is walletNameForCDP
      network: network,
      createdAt: new Date(),
      userIdentifier: assignedTo, // Storing who this wallet is assigned to
    };
    managedWalletsStore.push(newManagedWallet);
    console.log(`[API /wallets] Created: ID ${newManagedWallet.serviceId}, Addr: ${newManagedWallet.address}, Name: ${newManagedWallet.name}, AssignedTo: ${assignedTo}, Net: ${network}`);
```

```
    res.status(201).json(newManagedWallet);
  } catch (error: any) {
    console.error("[API /wallets] Error:", error.message);
    res.status(500).json({ error: "Failed to create general service wallet", details: error.message
});
  }
});

// In src/server.ts
app.post('/users/wallet', async (req, res) => {
  const { username, network = "base-sepolia" } = req.body; // 'username' from the dApp user

  if (!username || typeof username !== 'string' || username.trim() === '') {
    return res.status(400).json({ error: "'username' (string) is required and cannot be empty." });
  }
  if (typeof network !== 'string' || !network) {
    return res.status(400).json({ error: "Valid 'network' (string) is required."});
  }
  console.log(`[API /users/wallet] Request for username: ${username} on network ${network}`);

  // Use a combination of username and network for the unique name in CDP & our store
  const cdpWalletName = `DAppUser-${username}-${network}`;

  // In a production app, query your database for a wallet linked to this 'username' and 'network'.
  // For demo, we search by the 'name' field which we've constructed to be unique.
```

```
  let userManagedWallet = managedWalletsStore.find(
    w => w.name === cdpWalletName && w.network.toLowerCase() === network.toLowerCase()
  );

  if (userManagedWallet) {
    console.log(`[API /users/wallet] Found existing wallet for username ${username} (${cdpWalletName}): ${userManagedWallet.address}`);
    return res.status(200).json({
      message: "Existing wallet retrieved for user.",
      ...userManagedWallet
    });
  }

  // If not found, create one
  try {
    const cdp = getCdpClient();
    console.log(`[API /users/wallet] Creating new wallet for username ${username} with CDP name: ${cdpWalletName}`);

    const newCdpAccount = await cdp.evm.createAccount({ name: cdpWalletName });

    internalWalletIdCounter++;
    const newManagedWallet: ManagedWallet = {
      serviceId: `srv-userwallet-${internalWalletIdCounter}`,
      cdpAccountUuid: newCdpAccount.accountUuid,
```

```
      address: newCdpAccount.address,
      name: newCdpAccount.name, // This will be cdpWalletName
      network: network,
      createdAt: new Date(),
      userIdentifier: username, // Storing the dApp username
    };

    managedWalletsStore.push(newManagedWallet);
    console.log(`[API /users/wallet] New wallet created for username ${username}:
${newManagedWallet.address}, Name in CDP: ${newManagedWallet.name}`);

    res.status(201).json({
      message: "New wallet created successfully for user.",
      ...newManagedWallet
    });
  } catch (error: any) {
    console.error(`[API /users/wallet] Error for username ${username}:`, error.message);
    res.status(500).json({ error: "Failed to create or retrieve user wallet", details: error.message });
  }
});

// In src/server.ts
app.post('/wallets/:walletAddress/faucet', async (req, res) => {
  const walletAddressParam = req.params.walletAddress as string;
```

```
  const { network = "base-sepolia", token = "eth" } = req.body; // Only 'eth' supported by cdp.evm.requestFaucet


  console.log(`[API /faucet] Request for ${walletAddressParam} on ${network} to get ${token}`);

  if (!isAddress(walletAddressParam)) return res.status(400).json({ error: "Invalid wallet address."});

  if (token.toLowerCase() !== "eth") return res.status(400).json({ error: "CDP Faucet currently supports ETH only for EVM."});



  const managedWallet = managedWalletsStore.find(w => w.address.toLowerCase() === walletAddressParam.toLowerCase() && w.network.toLowerCase() === network.toLowerCase());

  if (!managedWallet) return res.status(404).json({ error: `Wallet ${walletAddressParam} on ${network} not managed by this service.`});


  try {

    const cdp = getCdpClient();

    const faucetResult = await cdp.evm.requestFaucet({

      address: managedWallet.address, // Address of the wallet to fund

      network: network, // e.g., "base-sepolia"

      token: token,     // "eth"

    });

    console.log(`[API /faucet] Faucet request for ${managedWallet.address} submitted. TxHash: ${faucetResult.transactionHash}`);


    console.log(`[API /faucet] Waiting for faucet tx ${faucetResult.transactionHash} confirmation...`);
```

```
    const receipt = await viemPublicClient.waitForTransactionReceipt({ hash:
faucetResult.transactionHash, timeout: 180_000 });



    if (receipt.status === 'success') {

      console.log(`[API /faucet] Faucet tx ${faucetResult.transactionHash} confirmed for
${managedWallet.address}.`);

      res.status(200).json({ message: "Faucet funds requested and confirmed.", transactionHash:
faucetResult.transactionHash, receiptStatus: receipt.status });

    } else {

      // This path might be less common if waitForTransactionReceipt throws on failure

      throw new Error(`Faucet transaction ${faucetResult.transactionHash} confirmed but reverted
(status: ${receipt.status}).`);

    }

  } catch (error: any) {

    console.error(`[API /faucet] Error for ${walletAddressParam}:`, error.message);

    // CDP SDK might throw specific error types for faucet limits

    if (error.message?.toLowerCase().includes('already requested faucet') ||
error.message?.toLowerCase().includes('limit exceeded')) {

        return res.status(429).json({ error: "Faucet funds likely already requested recently for this
address or rate limit hit.", details: error.message });

    }

    res.status(500).json({ error: "Failed to request or confirm faucet funds.", details:
error.message });

  }

});


// In src/server.ts (ensure sleep function is defined globally if not already)

function sleep(ms: number): Promise<void> {
```

```
  return new Promise((resolve) => setTimeout(resolve, ms));
}

app.post('/wallets/:walletAddress/send-batch-eth', async (req, res) => {
  const senderAddressParam = req.params.walletAddress as string;
  const { recipients, amountPerRecipient, network = "base-sepolia" } = req.body;

  console.log(`[API /send-batch-eth] Wallet ${senderAddressParam} on ${network}: send
${amountPerRecipient} ETH to ${recipients?.length} recipients.`);

  // --- Input Validations ---
  if (!isAddress(senderAddressParam)) return res.status(400).json({ error: "Invalid sender wallet
address." });
  if (!Array.isArray(recipients) || recipients.length === 0) return res.status(400).json({ error:
"Recipients must be a non-empty array."});
  if (!amountPerRecipient || typeof amountPerRecipient !== 'string' ||
parseFloat(amountPerRecipient) <= 0) {
    return res.status(400).json({ error: "Invalid amountPerRecipient."});
  }
  const validRecipients = recipients.filter(addr => typeof addr === 'string' && isAddress(addr));
  if (validRecipients.length !== recipients.length || validRecipients.length === 0) {
    return res.status(400).json({ error: 'Some/all recipient addresses invalid or none provided.' });
  }

  const managedSender = managedWalletsStore.find(w => w.address.toLowerCase() ===
senderAddressParam.toLowerCase() && w.network.toLowerCase() ===
network.toLowerCase());
```

```
  if (!managedSender) return res.status(403).json({ error: `Wallet ${senderAddressParam} on ${network} not managed by this service.` });


  const cdpSenderAddress = managedSender.address; // Use the address from our managed store

  let amountInWei: bigint;

  try {

    amountInWei = parseEther(amountPerRecipient as `${number}`);

    if (amountInWei <= 0n) throw new Error("Amount must be positive.");

  } catch (e) { return res.status(400).json({ error: `Invalid amount format: ${amountPerRecipient}.` }); }


  try {

    const cdp = getCdpClient();

    console.log(`[API /send-batch-eth] Batch send from ${cdpSenderAddress} for ${validRecipients.length} recipients...`);


    // This part mirrors the sendManyTransactions.ts example structure

    const transactionSubmissionPromises = validRecipients.map((recipient, index) => {

      return (async (currentIndex: number) => {

        let retryCount = 0; const MAX_RETRIES = 3; const BASE_DELAY = 1000;

        const currentRecipient = recipient as `0x${string}`;

        while (true) {

          try {

            const txResult = await cdp.evm.sendTransaction({

              address: cdpSenderAddress, // The EOA managed by our service
```

```
            network,

            transaction: { to: currentRecipient, value: amountInWei },

          });

          console.log(`[API /send-batch-eth] Submitted Tx #${currentIndex + 1} to
${currentRecipient}. Hash: ${txResult.transactionHash}`);

          return { txHash: txResult.transactionHash, index: currentIndex, recipient:
currentRecipient };

        } catch (error: any) {

          const isRateLimit = error.message?.toLowerCase().includes('rate limit') ||
error.message?.toLowerCase().includes('429'); // Basic check

          if (isRateLimit && retryCount < MAX_RETRIES) {

            retryCount++;

            const delay = BASE_DELAY * Math.pow(2, retryCount) + (Math.random() *
BASE_DELAY / 2);

            console.warn(`[API /send-batch-eth] Tx #${currentIndex + 1} to ${currentRecipient} rate
limited, retrying in ${Math.round(delay)}ms...`);

            await sleep(delay);

          } else {

            console.error(`[API /send-batch-eth] Failed to submit Tx #${currentIndex + 1} to
${currentRecipient}:`, error.message);

            throw { recipient: currentRecipient, error: error.message || 'SDK submission error',
index: currentIndex }; // Throw structured error

          }

        }

      }

    })(index);

  });
```

```
    const submissionResults = await Promise.allSettled(transactionSubmissionPromises);

    console.log(`[API /send-batch-eth] All ${validRecipients.length} transaction submissions attempted.`);



    const receiptPromises = submissionResults.map(async (settledResult, index) => {

      const recipient = validRecipients[index];

      if (settledResult.status === 'rejected') {

        const reason = settledResult.reason as any;

        console.error(`[API /send-batch-eth] Tx #${index + 1} to ${recipient} submission ultimately failed: ${reason.error || String(reason)}`);

        return { recipient, status: 'failed_submission' as const, error: reason.error || String(reason), index, txHash: undefined, blockNumber: null };

      }



      const { txHash } = settledResult.value;

      console.log(`[API /send-batch-eth] Tx #${index + 1} (Hash: ${txHash}) to ${recipient}. Waiting for receipt...`);

      try {

        const receipt = await viemPublicClient.waitForTransactionReceipt({ hash: txHash, timeout: 60_000 });

        console.log(`[API /send-batch-eth] Tx #${index + 1} to ${recipient} confirmed! Status: ${receipt.status}`);

        return { recipient, status: receipt.status, txHash, index, blockNumber: receipt.blockNumber, error: receipt.status === 'reverted' ? 'Transaction reverted' : null };

      } catch (receiptError: any) {

        console.error(`[API /send-batch-eth] Tx #${index + 1} (Hash: ${txHash}) to ${recipient} timed out on receipt:`, receiptError.message);

        return { recipient, status: 'timeout_receipt' as const, txHash, index, error: receiptError.message || 'Timeout on receipt', blockNumber: null };
```

```
      }
    });

    const finalResults = await Promise.all(receiptPromises);
    const confirmedCount = finalResults.filter(r => r.status === 'success').length;
    const failedCount = finalResults.length - confirmedCount;

    console.log(`[API /send-batch-eth] Batch confirmation complete. Confirmed: ${confirmedCount}, Failed/Other: ${failedCount}`);
    res.status(200).json({
      message: `Batch ETH send processed for ${validRecipients.length} recipients. Confirmed: ${confirmedCount}, Failed/Other: ${failedCount}.`,
      submittedCount: validRecipients.length,
      confirmedCount,
      failedCount,
      results: finalResults
    });
  } catch (error: any) {
    console.error("[API /send-batch-eth] Critical error in batch process:", error.message);
    res.status(500).json({ error: error.message || 'Internal Server Error during batch ETH send.' });
  }
});
```