# Yield Infra for NeoBanks

Raga Finance is building a **yield infrastructure layer for Web3 neobanks**, enabling any neobank to instantly offer DeFi investment products to its users across multiple chains and protocols. Through a unified **SDK and API suite**, neobanks gain access to a standardized interface that abstracts away the complexity of multi-chain smart contracts, yield strategies, data normalization, risk evaluation, and transaction execution.

Neobanks can seamlessly

- discover high quality yield opportunities,
- view historical performance and risk analytics,
- deposit or withdraw into a single vault that manages all the strategies
- add and change strategies according to markets and curators
- see real time PnL and portfolio performance

Users can

- Invest in yield optimised vaults
- Withdraw money anytime they want
- See daily PnL of the portfolio in dollar value

This system is designed to be **protocol agnostic**, **non-custodial**, and **scalable**, enabling integration of any future DeFi protocol or chain with minimal engineering effort.

## Technical Architecture

### Requirements

The yield infrastructure must be easy to integrate using API keys and a lightweight SDK. Neobanks should be able to:

#### Core Functional Requirements

1. Fetch up-to-date information about supported DeFi protocols
   - APY, reward APY
   - TVL, liquidity data
   - Supported networks
   - Risk scores
2. Fetch yield strategies across lending, staking, LPing, restaking, RWA, or vault strategies
3. Retrieve detailed information about a specific strategy
4. Generate transaction payloads for deposit and withdraw

5. Execute transactions through users' non-custodial wallets or the neobank's signing system
6. Fetch real time protocol performance data including
    - Historical APY charts
    - TVL history
    - Volatility and risk indicators
7. Provide analytics for risk assessment across strategies
8. Show user specific portfolio data
    - Positions
    - Current value
    - Accrued yield
    - Claimable rewards
9. Handle cross-chain and multi-protocol interactions
10. Provide unified data formatting across protocols (normalizing APY, asset metadata, decimals, network IDs)

## Additional Requirements

11. Provide market benchmarks (average stablecoin APY, lending averages)
12. Provide webhook or callback support for transaction confirmations
13. Caching layer to improve response times for commonly accessed data
14. Real time monitoring of protocol parameter changes (APY updates, reward rate changes)
15. Unified error handling and simulation endpoints
16. Should support future chains and protocols through simple backend configuration

## Relevant Protocols to be integrated v1:

1. Aave
2. Morpho
3. Pendle
4. Euler
5. Spectra
6. Lido

## Relevant chains for v1:

1. Ethereum
2. Base
3. BSC
4. Arbitrum

Below are the architecture for building SDK and APIs for yield infra

## Backend

The backend is the core engine powering the yield infrastructure. It stores protocol data, aggregates analytics, normalizes yields across chains, and prepares transaction payloads. It ensures the frontend and SDK receive consistent, structured, real time information regardless of protocol differences through APIs

Backend has following responsibilities:

1. **Protocol Data Aggregation Layer:** Continuously fetches, scrapes, indexes, or subscribes to every integrated DeFi protocol and stores:
   - Real APY
   - Reward APY (token emissions, distribution rates)
   - TVL and liquidity depth
   - Borrow/supply rates (if applicable)
   - Utilization ratio
   - Smart contract addresses
   - Supported networks
   - Token metadata (address, decimals, symbol)
   - Historic time-series (APY, TVL, yield, price)
   - Protocol status and risk metrics
   - Oracle feeds or price data

   Scrapers run at configurable intervals depending on protocol volatility.

2. **Strategy Normalization Layer:** Different protocols expose yields differently (supply APY, reward APR, LP APR, trading fees, rebasing yield, staking APR). This layer:
   - Normalizes APY/APR across all protocols
   - Converts strategy metadata into a unified format
   - Generates risk flags or scores based on protocol data
   - Computes aggregated metrics like blended APY, net yield, historical volatility

   This ensures that neobanks always see standardized, comparable metrics.

3. **Portfolio Tracking and Accounting Engine:** This engine reconstructs positions using on-chain events, ensuring accurate accounting even if transactions occur outside the neobank UI. Maintains user-specific state by tracking:
   - Deposits into any protocol
   - Withdrawals (completed, pending, failed)
   - Real time value of positions
   - Accrued yield and claimable rewards
   - Cost basis
   - Cross chain balances
   - PnL computation over time

This engine reconstructs positions using on-chain events, ensuring accurate accounting even if transactions occur outside the neobank UI.

4. **Risk & Analytics Module**: Provides deeper analytics for neobanks:
    - Historical APY volatility
    - Liquidity and slippage modeling
    - Protocol safety flags
    - Smart contract audits metadata
    - Risk scores using configurable parameters
    - Benchmark APY and cross market comparisons

    This enables neobanks to offer richer insights and transparency to users.

5. **API Orchestration Layer:** Serves all the API endpoints:
    - Protocol endpoints (strategies, protocol info)
    - Portfolio endpoints
    - Recommendations endpoints
    - Historical data endpoints
    - Market data endpoints

    Handles:

    - Authentication
    - Rate limiting
    - Caching
    - Schema validation
    - Unified JSON responses

6. **Chain Indexing & Node Infrastructure:** For high performance multi-chain operations, backend uses:
    - Dedicated RPC nodes
    - Indexers for each chain
    - Event listeners for protocol contract logs
    - Price oracles
    - Internal caching and message queues for real time updates

    This ensures the infra scales with the number of chains and protocols integrated.

## Vaults

Vaults are needed for distribution of funds to relevant strategies and to combine APY of all.
We will use Veda's Boring Vault as a base and develop strategies on top.
**Working**

1. Neobank will select what all strategies it wants to integrate into their app from the front end.

2. They'll add a curator or select %age allocation in each strategy
3. A fee will be set for all the vaults and will be sent to fee vault in regular interval
4. Once they have selected everything, contracts and a backend bot will be deployed.
5. Neobank can see the vault address and API key to integrate the vaults in their frontend

## Execution Engine

Execution Engine is required to allocate funds in their relevant strategies. It'll be written in golang and run in a TEE to prevent exposure of keys.
**Key Functionalities**

1. Should be able to allocate funds in different strategies
2. Should be able to withdraw funds from different strategies
3. Should be able to balance withdrawals and deposits effectively
4. Should follow the allocation of funds decided by the curator of the vault

## API

Below we outline the major API endpoints, including their method, purpose, inputs, outputs, and typical use cases. These endpoints cover fetching available yield strategies, retrieving details of specific protocols or pools, executing transactions, and obtaining user portfolio data.

### Frontend(like conduit)

#### Non Authentication(anyone can view)

**GET /getAllStrategies**

**Description:** Retrieves all of available strategies (e.g. lending pools or interest-bearing vaults on protocols like Aave, Compound, etc.). The response includes key info for each strategy: protocol name, underlying asset, current APY (interest rate), reward APY (if rewards are paid in another token), total value locked (TVL), supported network, and an identifier for the strategy.
**Response:**

```
[

 {

   "strategyIdentifier":"fgjhkl",

   "strategyName":"Raga Hype Delta Neutral",

   "strategyAddress":"0x042EE5a5Ef98C3cefEACFa2037354834613e2cad",

   "strategyType":"Lending/DEX/Delta Neutral/Liquid staking",
   "APR":100,

   "TVL":1000000,

 }

]
```

**POST /getStrategies**

**Description:** Fetches detailed information about a specific yield **protocol or strategy**. This can be used to get in-depth data on a particular pool/vault. For instance, if a user selects a **single yield opportunity** (say, the DAI lending pool on Compound), this endpoint returns comprehensive details about that pool.

**Inputs:**

```
["identifier_1","identifier_2","identifier_3","identifier_4"]
```

**Response:**

```
{

 "identifier_1":{

   "strategyName":"Raga Hype Delta Neutral",

   "depositAssetAddress":"0x00",

   "depositAssetName":"USDC",

   "description":"Raga Hype Delta Neutral is a strategy that uses a delta-neutral
approach to trade the Hype token on Hyperliquid.",

   "strategyAddress":"0x042EE5a5Ef98C3cefEACFa2037354834613e2cad",

   "strategyType":"Lending/DEX/Delta Neutral/Liquid staking",

   "APR":100,

   "TVL":1000000,

   "RISK_PROFILE":"Low",

   "APR_CHART":{
     "timestamp_1": "value_1",

     "timestamp_2": "value_2",

   },

   "TVL_CHART":{

     "timestamp_1": "value_1",

     "timestamp_2": "value_2",

   }

 }

  "Identifier_2":{

 }

}
```

## Authentication(After logging in)

**POST /getUser**

**Description:** Provides user information coming to the platform and connecting his mail, wallet or any form of authentication. It'll return basic information about the user

**Input:**

```
{

  "user": "user_identifier",

}
```

**Output:**

```
{
 "user_identifier":{
   "Name": "John Doe",
   "Email": "john.doe@example.com",
   "Phone": "1234567890",
   "Address": "123 Main St, Anytown, USA",
   "verified": true,
 }
}
```

**POST /createUser**

**Description:** If getUser gives null value, then create a user and identifier and return basic information about the user

**Input:**

```
{

 "Name": "John Doe",

 "Email": "john.doe@example.com",

 "Phone": "1234567890",

 "Address": "123 Main St, Anytown, USA",

 "verified": false,

}
```

**Output:**

```
{

 "user_identifier":

 {

   "Name": "John Doe",

   "Email": "john.doe@example.com",

   "Phone": "1234567890",

   "Address": "123 Main St, Anytown, USA",

   "verified": true,

 }
}
```

**POST /createVault**

**Description:** Users can create vaults with their desired allocation to different protocols on a specific chain. They can select between different DeFi protocols present on the chain depending on the risk metrics, strategy type and personal preference.

**Inputs:**

```
{

 "chain_id": 1,

 "strategy_identifier_1": "strategy_allocation_1",

 "strategy_identifier_2": "strategy_allocation_2",

 "strategy_identifier_3": "strategy_allocation_3",

 .

 .

 .

}
```

**Response:**

api_key=this is user specific. A single user has a single api key

```
{

 "status": "success | inprocess | failed",

 "vault_address": "0x1234567890123456789012345678901234567890",

 "vault_identifier": "0x1234567890123456789012345678901234567890",

 "chain_id": "1",

 "bot_address": "0x1234567890123456789012345678901234567890",

 "api_key": "1234567890",

}
```

**POST /getUserVaults**

**Description:** Provides a consolidated view of a **Neobanks's yield portfolio** across all supported protocols and chains. This endpoint aggregates the user's positions (deposits and earned yields) in one place.
**Inputs:**

```
{

 "user": "user_identifier",

}
```

**Response:**

```
{

 "user_identifier":{

   "api_key": 12345,

   "testnet":[{

     "vault_address": "0x1234567890123456789012345678901234567890",

     "active": "true",

     "vault_name": "Vault 1",

     "vault_description": "Vault 1 description",

     "vault_type": "Vault 1 type",

     "vault_risk_profile": "Vault 1 risk profile",

     "vault_apy": 100,

     "vault_tvl": 1000000,

     "vault_apy_chart":{

       "timestamp_1": "value_1",

       "timestamp_2": "value_2",
     },

     "vault_tvl_chart":{

       "timestamp_1": "value_1",

       "timestamp_2": "value_2",

     }

   }]

   "mainnet":[]

 }

}
```

**POST /updateVaults**

**Description:** Users can update the vault allocation, add or remove strategy.
**Inputs:**

```
{

 "vault_identifier": "",

 "chain_id": "",

 "allocations": {

   "strategy_identifier_1": "strategy_allocation_1",

   "strategy_identifier_2": "strategy_allocation_2",

   "strategy_identifier_3": "strategy_allocation_3",

 }

}
```

## Authentication(Neobank user calls)

**POST /getVaults**

**Description:** Lists all the vaults that are active for neobanks
**Input**:

```
{
 "api_key": "wgueydbfhcj",
}
```

**Output:**

```
{
   "testnet":[{
     "vault_address": "0x1234567890123456789012345678901234567890",
     "active": "true",
     "vault_name": "Vault 1",
     "vault_description": "Vault 1 description",
     "vault_type": "Vault 1 type",
     "vault_risk_profile": "Vault 1 risk profile",
     "vault_apy": 100,
     "vault_tvl": 1000000,
     "vault_apy_chart":{
       "timestamp_1": "value_1",
       "timestamp_2": "value_2",
     },
     "vault_tvl_chart":{
       "timestamp_1": "value_1",
       "timestamp_2": "value_2",
     }
   }],

   "mainnet":[]

}
```

**POST /getUserPortfolio**

**Description:** List all the deposits made by user on neobanks in different vaults

**Input:**

```
{

 "user_address": "0x9ub",

}
```

**Output:**

```
{
 "user_address": "0x1234567890123456789012345678901234567890",
 "deposits":{
   "vault_identifier_1":{
     "vault_address": "0x1234567890123456789012345678901234567890",
     "user_shares": "124111",
     "average_buy_price": "13443",
     "last_buy_time": "1345",
     "vault_risk_profile": "Vault 1 risk profile",
     "vault_apy": 100,
     "vault_tvl": 1000000,
   }
   "vault_identifier_2":{
     "vault_address": "0x1234567890123456789012345678901234567890",
     "user_shares": "124111",
     "average_buy_price": "13443",
     "last_buy_time": "1345",
     "vault_risk_profile": "Vault 1 risk profile",
     "vault_apy": 100,
     "vault_tvl": 1000000,
   }
   .
   .
   .
 }
 "withdrawals":{
   "vault_identifier_1":{
     "vault_address": "0x1234567890123456789012345678901234567890",
     "withdrawal_shares": "124111",
     "average_sell_price": "13443",
     "withdrawal_time": "1345",
   }
   "vault_identifier_2":{
     "vault_address": "0x1234567890123456789012345678901234567890",
     "withdrawal_shares": "124111",
     "average_sell_price": "13443",
     "withdrawal_time": "1345",
   }
   .
   .
   .
 }
}
```

**Post /getVaultDetails**

## SDK Structure and Components

The SDK is structured into several key modules and components, each responsible for a part of the functionality:

- **API Client Module:** This is the core of the SDK that wraps each API endpoint into a clean function call. Developers can call methods like `sdk.getVaults()` or `sdk.getUserPortfolio(userAddress)` directly, without constructing HTTP requests. The client module manages the low-level details: attaching the API key, routing requests to the correct environment (testnet vs mainnet if applicable), and handling errors or retries. It ensures a consistent interface in the programming language of choice, abstracting the REST endpoints behind simple function calls.

- **Authentication & Config:** The SDK provides a straightforward way to configure credentials and environment. For instance, upon initialization, the developer supplies the API key (and optionally selects a network or environment). The SDK securely stores this (often just in memory/config, since each API call will include the key in headers). It may also allow configuration of default parameters (e.g. default network or default user context) to avoid repeating them in every call.

- **State Management:** To improve performance and provide a smoother UX, the SDK handles some client-side state. This includes caching of data and tracking of user-specific state. It can also maintain the state of the user's current portfolio in memory; so if the app requests the portfolio data, it can instantly provide the last-known data while fetching updates in the background. The state management module might use observables or event emitters to notify the app of updates (e.g. "new data fetched" or "transaction status changed"), which the frontend can subscribe to.

- **Transaction Manager:** This component of the SDK streamlines the process of executing transactions through the platform. The contracts after being implemented would need an interface to deposit and withdraw and this component will greatly enable that from the neobanks frontend. it could hand off the transaction to the neobank's secure signing module. The transaction manager also can poll or listen for the transaction receipt and update the portfolio state once the transaction is confirmed. All of this logic being in the SDK means the neobank's developers do not have to write custom code for each protocol's transaction flow; it's handled in a unified way.

- **Utility and Helper Functions:** The SDK includes various helper utilities to make building the frontend easier. This can include formatting helpers (e.g. to format APY percentages or fiat values consistently), asset conversion tools (perhaps converting between token units and human-readable units), and network utilities (for instance, opening a block explorer link for a transaction hash). These utilities enforce consistent behavior across the app – for example, ensuring all percentages are rounded the same way or all addresses are checksummed.

- **UI Component Library (Optional):** To accelerate development, the SDK may offer a set of pre-built UI components or templates for common interfaces in the yield app. Each component encapsulates a piece of frontend logic and design, using the SDK under the hood to fetch data. The goal is to provide a white-label frontend experience that neobanks can drop into their applications and then customize as needed, rather than starting from scratch. If the SDK doesn't include literal UI components, it would at least come with reference implementations or example code for building these screens with the SDK's data, serving as a blueprint for integration.

Overall, the SDK is designed with modularity in mind - each piece (data fetching, state mgmt, transactions, UI helpers) is decoupled, so neobanks can choose how much of the stack to use. For instance, some may use only the API client and build a completely custom UI, while others might leverage the provided UI components to speed up development.

## Frontend Integration and Workflow

The SDK greatly simplifies how the neobank's frontend (or backend) integrates with the yield infrastructure. Here's a typical workflow and frontend structure using the SDK:

- **Initialization:** The neobank's app initializes the SDK early in the app lifecycle (for example, during app startup). The API key is provided here, and any config like default network or enabling test mode is set. After initialization, the SDK is ready to be used across the app.

- **Earn screen:** The app uses the SDK to fetch and display earn page components. This screen allows users to browse available protocols and yields, potentially with filtering options. The SDK abstracts the complexity of querying multiple protocols – the front-end just sees a unified list.

- **Strategy Details Page:** When a user selects to deposit on Earn page, the app navigates to a detail page. On this page, the SDK can be used to fetch additional details The SDK provides the data needed to show metrics like historical APY charts, risk scores, contract addresses, etc. The front-end might also subscribe to SDK state here.

- **Portfolio Page:** The neobank's app likely has a "Portfolio" or "My Earnings" section. This page would use `sdk.getPortfolio(user)` on load, or possibly subscribe to a portfolio store provided by the SDK. The SDK returns the user's current positions and earnings across all strategies. The front-end displays this in a consolidated view, using tables or cards. If a UI component is provided, the developer can use it directly to show a formatted summary of the user's investments. This real-time position tracking is made easy by the SDK's ability to fetch across protocols and update as on-chain data changes.

- **Transaction Flow (Deposit/Withdraw):** When a user decides to invest (deposit) or withdraw, the front-end triggers the SDK's transaction helper. The SDK takes care of integrating with the wallet. If the neobank uses a custodial key management, the SDK could instead call a callback in the host app to perform the signing. After the user confirms, the SDK can monitor the transaction status. The UI can leverage SDK events to show a loading spinner or confirmation message. This process is consistent for any protocol.

- **Updates and Notifications:** As the user continues to use the app, the SDK can listen for certain updates. For example, if new yield strategies become available or if a protocol's rates change significantly, the SDK might emit an event or the app can call refresh methods. The neobank could also run background refreshes (perhaps calling `sdk.getPortfolio` every minute or using webhooks if supported) to keep data fresh. The SDK ensures that such updates are handled efficiently (using caching or only fetching deltas) so the app remains responsive.

In terms of **frontend structure**, neobanks can integrate this SDK either directly in their existing app or as a separate module. A generic implementation could involve adding a new section to the app (often labeled "Earn", "DeFi", or "Invest") which is powered by the SDK. Within that section, the screens and flow described above would reside. The neobank can follow a reference implementation provided with the SDK, which outlines the UI flow and how SDK calls map to user actions. Because the SDK provides a lot of the heavy lifting, adding this new module to the app does not require deep blockchain expertise on the neobank's team – they leverage the SDK's abstractions and focus on design and integration.

## Customization and Extensibility for Neobanks

One of the core design goals of the SDK and API is to be **flexible and customizable** so that each neobank can tailor the user experience to its brand and user needs:

- **UI Customization:** If the SDK includes pre-built UI components, these are built to be themable and adaptable. Neobanks can apply their own branding (colors, fonts, logos) to these components easily, ensuring the new DeFi yield features feel native to their app's look and feel. The SDK does not enforce any specific look, it only provides the data and actions.

- **Selective Feature Enablement:** The infrastructure is **protocol-agnostic** and supports many protocols, but a neobank might choose to enable only a subset. The SDK/API can be configured to **whitelist** certain protocols, assets, or networks (likely through API query parameters or initialization settings). For instance, a neobank focusing on low-risk yields might only enable stablecoin lending protocols and disable more experimental ones. This ensures compliance or alignment with the neobank's risk policies.

- **Extensibility:** If the neobank wants to add additional logic or data, the SDK is designed to accommodate that. The SDK might allow injection of custom data sources or plugin modules. While the default SDK covers the common needs, advanced users can extend classes or use the API client directly for custom requests. The API's rich set of endpoints (e.g. historical data, benchmarks) means developers can create custom analytics or displays beyond the standard ones.

Competitor and inspiration: [https://docs.vaults.fyi/](https://docs.vaults.fyi/)