ChatGPT

Integrating Cryptocurrency Wallet Functionality into a Modern FinOps Platform

Introduction: This report explores how a modern financial operating system (like the Obsidian FinOps platform) can evolve to support cryptocurrency wallet features. We cover the required **infrastructure, protocols, and compliance frameworks** for both custodial and non-custodial crypto wallets, and evaluate **practical implementation options** (from using third-party custodial services or WalletConnect to embedding Web3 wallet SDKs). We then discuss how to **integrate crypto balances and transactions** into an account-based dashboard, best practices for **secure key management** (private key storage, enclaves, HSMs, recovery), and the role of **smart contract wallets (EIP-4337/account abstraction)** in this context. Further sections outline **UI/UX requirements** for onboarding and transacting with crypto, specific **Brazilian legal/tax compliance** considerations (IRPF reporting, Receita Federal rules, LGPD), and a comparison of the **trade-offs between architectural approaches** (embedded self-custody vs. wallet-as-a-service vs. running full blockchain nodes). Finally, we examine how the platform's existing security model (change-set ledger, policy-as-code, kill-switches, human-in-loop approvals) can **extend to crypto flows**, ensuring that agent-driven blockchain transactions remain controlled and auditable.

1. Infrastructure & Protocols for Operating a Crypto Wallet

Custodial vs. Non-Custodial Wallet Infrastructure: At a high level, the infrastructure differs markedly depending on whether the platform will custody user assets or not. A **non-custodial wallet** is essentially a client-side tool – it helps users generate and store their own keys (typically via a browser or mobile app) and interact with blockchains, but the platform never holds the assets itself [1] [2]. Non-custodial wallets like MetaMask or Trust Wallet simply provide interfaces; no special licenses are required for the provider since users retain control of their crypto keys [1]. In contrast, a **custodial wallet** means the platform (or a third-party service on its behalf) holds private keys and assets for users. This requires a robust backend wallet infrastructure – typically secure key storage (in databases, Hardware Security Modules, or MPC networks), blockchain nodes or API connections for each supported network, and processes to handle deposits, withdrawals, and transaction signing. Custodial providers must meet strict requirements: obtaining appropriate licenses, implementing KYC/AML identity checks, appointing compliance officers, and adhering to cybersecurity standards [3]. In many jurisdictions, custodial wallet operators are classified as **Virtual Asset Service Providers (VASPs)** and subject to financial regulations. For example, in the U.S. a custodial wallet business might need a Money Transmitter License [3], while in Brazil the new Law No. 14.478/2022 mandates **Central Bank authorization** for VASPs [4]. Non-custodial wallets largely fall outside such requirements since they *"do not hold or transmit user funds"*, and generally are not considered MSBs or VASPs [5] [6] (though they must still follow data protection laws like LGPD).

Blockchain Network Connectivity & Protocols: Whether custodial or not, the platform needs connectivity to blockchain networks. The core protocols involved include blockchain node APIs (e.g. Ethereum's JSON-RPC interface for sending transactions and querying balances) and standards for key management. A wallet typically adheres to key-generation standards like **BIP-39** (mnemonic seed phrases) and **BIP-44** (HD wallet derivation paths) to generate addresses on various chains. For Ethereum and other EVM chains, the

platform might use libraries implementing **EIP-155** (chain IDs) and **EIP-1193** (the web3 provider interface) if integrating with dApps. Operating a wallet also implies supporting protocols for **transaction creation, signing, and broadcast** on each chain: for example, constructing Ethereum transactions (RLP encoded) and broadcasting via an Ethereum node or an RPC provider. If multiple blockchains are supported (BTC, Ethereum, Polygon, etc.), the infrastructure may involve either running full nodes for each network or leveraging a multi-chain API service. Many providers (Alchemy, Infura, QuickNode, Blockdaemon, etc.) offer scalable access to blockchain data and broadcasting. For instance, services like Covalent's API can fetch *"multichain ERC20, NFT and native token balances and transfers"* across 100+ networks [7], and provide decoded transaction history for addresses (useful for building wallet activity feeds and tax tools) [8]. These APIs allow the platform to integrate chain data without running its own node farm, though running a full node can be considered for trust and reliability (especially for Bitcoin or if needing direct mempool monitoring).

Compliance Frameworks: Operating a crypto wallet introduces compliance considerations on multiple fronts. **KYC/AML (Anti-Money Laundering)** procedures are essential for custodial wallets – users will need to verify their identity before they can deposit or withdraw crypto. The platform would have to implement onboarding workflows to collect IDs, CPF (for Brazil), etc., and employ AML transaction monitoring. Many countries (including Brazil) are implementing the **FATF Travel Rule**, which requires that identifying information of sender/receiver accompanies crypto transfers over a certain amount [9]. A custodial wallet infrastructure should be ready to integrate Travel Rule solutions (e.g. Fireblocks integrates with Notabene for secure transmission of required customer data on transfers) [10]. Additionally, custodial providers are expected to screen wallet addresses against sanctions and blacklist databases (to avoid facilitating transfers to sanctioned entities). On the other hand, non-custodial wallet software typically does not enforce KYC by itself (users can generate a wallet anonymously), but if the platform's business model involves exchanging crypto to fiat or other regulated services, it may still require users to identify themselves at those touchpoints. In Brazil, the regulatory framework is evolving: as of 2024, the Central Bank (BCB) is phasing in oversight of crypto service providers with new licensing and reporting rules [4] [11]. Custodial wallet operators will need a BCB authorization and will have to implement AML policies consistent with those expected of financial institutions (internal controls, suspicious activity reporting, etc.). **Data protection** under LGPD is also critical – any personal data collected (e.g. identity documents, blockchain addresses linked to individuals, transaction history) must be stored and processed in compliance with Brazil's privacy law. The Obsidian FinOps architecture already emphasizes LGPD compliance (user consent, data deletion on request, audit logs) [12], so extending this to crypto means ensuring that any new personal data (like KYC info or behavioral data from crypto transactions) is properly safeguarded and only used with consent. All access to sensitive info and fund movements should be logged in tamper-evident audit trails for accountability [13].

In summary, a non-custodial approach minimizes regulatory overhead (no custody of funds, so generally no license or KYC needed for just providing the wallet interface [1]), but puts more responsibility on users for security. A custodial approach demands a heavier infrastructure and strict compliance program, essentially transforming the platform into a financial services provider subject to oversight. Many modern wallet offerings actually blend aspects of both via MPC or smart-contract wallets (giving users some control while a service co-manages keys); these hybrids can improve user experience but still require careful alignment with regulatory definitions.

2. Simplest Compliant Implementation Approaches

If the goal is to add crypto wallet features quickly and with minimal compliance risk, leveraging existing services and SDKs is often the best route. Below we outline a few implementation alternatives:

- **Use a Custodial Wallet-as-a-Service Provider:** Several companies offer APIs for fully managed crypto wallets, where they handle key management, blockchain connectivity, and compliance. Integrating such a custodial service means the platform outsources the heavy lifting. For example, **Fireblocks** and **BitGo** provide enterprise-grade custody platforms: the platform can use their API to create user wallets, initiate transfers, etc., while the provider ensures security (MPC key shards, HSM storage) and often carries insurance and compliance certifications. Custodial API providers typically have built-in features like address whitelists, policy controls, and even Travel Rule support (Fireblocks has a compliance suite for transaction monitoring and Notabene Travel Rule integration) [10]. The upside is quicker deployment of wallet functionality and reliance on an experienced custodian (which can ease regulator comfort). The downside is that the platform must perform KYC on its users (since they will be transacting through a custodial service) and possibly obtain a VASP license or partner with a licensed entity. In Brazil, a partnership with a locally licensed exchange or custodian could be an option – for instance, integrating with an exchange's custody solution so that compliance (reporting transactions to Receita Federal, etc.) is handled in that pipeline.

- **Non-Custodial via WalletConnect or External Wallets:** The arguably simplest approach from a compliance perspective is not to hold keys at all, but rather allow users to connect their own external crypto wallets. Using **WalletConnect**, the platform's app can interface with any user-held wallet (mobile wallets, MetaMask, browser extensions) in a secure way. The platform would generate a WalletConnect QR code/deeplink; the user approves connection and later approves any transaction on their personal device. This way, private keys never touch the platform's servers – eliminating custody risk and regulatory burden (since it's essentially acting as a software interface). WalletConnect integration is relatively straightforward (libraries exist for web and mobile that handle the pairing and relay) and would let users use familiar wallets. The trade-off is a *slightly* more fragmented UX: users must have an external wallet app and switch context to approve actions. However, this method means the platform can focus on features (like budgeting or analytics) without becoming a regulated custodian. Many fintech apps choose this route initially: e.g. allowing users to link a MetaMask to view balances or even execute trades via the app's interface, but all transactions still require the user's wallet confirmation. **Compliance-wise**, because the user is self-custodying, the platform can avoid onboarding friction (no KYC for pure wallet usage, though if fiat ramps or swaps are offered, those would trigger KYC via a partner). It's worth noting that even non-custodial platforms should include warnings about responsibility (users must keep their wallet secure) and perhaps provide education on Brazilian tax obligations for self-custodied crypto (since even if the platform doesn't report, the user still must).

- **Embed a Third-Party Wallet SDK (MPC/Social Login solutions):** A middle ground between pure external wallets and full custody is using an **embedded wallet SDK** that creates non-custodial wallets for users with a simplified experience. For example, **Coinbase Wallet-as-a-Service (WaaS)** and **Web3Auth** are services that let you spin up wallets where keys are partially managed by the provider and the user (through familiar login methods). Coinbase's WaaS provides an SDK to create on-chain wallets without seed phrases: it uses **multi-party computation (MPC)** such that the key is split between the user's device and Coinbase's servers [14]. This avoids the user having to record a

24-word seed and reportedly *"solves the key loss problem"* by offering backup via the MPC shares [15]. The platform can embed this and allow users to simply sign up with e.g. an email or OAuth, and behind the scenes an on-chain wallet is created for them. Coinbase manages the security and offers a **robust backup functionality to ensure users maintain access to assets** [15]. Importantly, Coinbase's solution is **non-custodial in spirit** (users can export or take control of the wallet), yet Coinbase as a reputable entity provides an additional safety net. From a compliance standpoint, an MPC wallet might not be *legally* custodial if the user's share of the key is required for any transaction (meaning the provider alone cannot move funds). This could reduce licensing burden, though this area is nuanced legally. Meanwhile, **Web3Auth** offers a social-login-based non-custodial wallet SDK: users can sign in with Google, Apple, etc., and it will generate a key that is split between the user and Web3Auth network (leveraging threshold cryptography). It touts *"non-custodial MPC wallet management"* with easy social logins [16]. For the platform, integrating Web3Auth is as easy as adding their SDK and configuring OAuth – in just a few lines of code developers can provide a user a wallet with, say, their Google account as the recovery method. This dramatically simplifies UX (no seed phrase) while keeping keys client-side (encrypted and sharded). Web3Auth and similar solutions do not perform KYC by default (since anyone with a Google account can get a wallet), so they're *compliance-light*. The platform would need to add KYC only if users are going to link to fiat or if certain transaction thresholds require it. **Thirdweb's Wallet SDK** is another notable example: Thirdweb provides an end-to-end toolkit for embedding wallets, including options for email/social logins, device-based local keys, and even smart contract wallets (ERC-4337) under the hood [17] [18]. Thirdweb's approach is non-custodial – *"users maintain full control over their digital assets"* with local or smart wallets [19] – but it gives developers plug-and-play UI components and even a **guest mode** to create a wallet for a new user silently which can later be linked to an email/password [20] [21]. Such SDKs handle a lot of edge cases (secure key storage on device, optional backup, etc.) and can drastically reduce development time.

- **Hybrid Custody via Licensed Partners:** If the platform wants to enable crypto without itself becoming a custodian, partnering with an existing licensed exchange or wallet provider can be effective. For example, the platform could integrate with an exchange's APIs so that when a user "creates a wallet" it is actually opening an account for them at that exchange (with the exchange holding the crypto). Many Brazilian exchanges (Mercado Bitcoin, Foxbit, etc.) or international ones operate in Brazil and have the required licenses. The platform's UI would act as a front-end, while behind the scenes the crypto is held by the partner. This requires strong integration work and sharing KYC data (the user would have to comply with the partner's KYC process), but it frees the platform from directly managing private keys or blockchain operations. The Obsidian FinOps could, for instance, let a user connect their Binance or Coinbase account via API to import balances and initiate crypto transactions – effectively treating those accounts as "wallets" in the FinOps dashboard. This approach is more for trading/investing use cases, whereas our primary focus is on making the FinOps an actual wallet; however, it's worth noting as an option for compliance (since the partner handles reporting to regulators).

Each approach above has pros and cons. To summarize the differences, we include a table comparing some major wallet integration options:

Comparison of Major Crypto Wallet API/SDK Options

Solution	Custody Model	Integration Effort	Features & Capabilities	Compliance/ Security Posture
Fireblocks (API)	Custodial (MPC-based) Platform holds keys via Fireblocks service	High – Enterprise API integration; back-end services needed	*Institutional-grade custody:* MPC key shards, policy engine (whitelisted addresses, spending limits), supports 30+ blockchains and tokens. Offers DeFi, staking, and token transfer APIs. *Developer tools:* SDKs in multiple languages, transaction simulators, and sandbox environment [22] [23].	**Regulated-facing:** Used by banks & exchanges – SOC 2 Type II certified. Provides built-in AML tools (address risk scoring, Travel Rule via Notabene) [10]. Fireblocks allows admin controls like wallet disable/enable (kill-switch) [24]. The business still needs KYC/AML programs, but Fireblocks offers compliance integrations.
Coinbase WaaS	Semi-custodial MPC (User and Coinbase share keys)	Medium – REST/ GraphQL API and SDKs; Coinbase Cloud account needed	*Embedded user wallets:* Create on-chain Ethereum wallets for users without seed phrases. Uses MPC such that keys can be secured by device and cloud—users can export keys. *User experience:* Supports user login via OAuth (e.g. Google) and device biometrics for transaction approvals [25] [26]. Scales to millions of wallets.	**Coinbase security:** Keys split so no single point of failure [14]. Coinbase is a publicly listed, US-regulated company, which can inspire trust. They promise **"robust, user-friendly backup"** and recovery options [27]. Compliance-wise, Coinbase WaaS itself doesn't enforce KYC for wallet creation (it's like a tech provider), but using it within a fintech app may require the fintech to ensure compliance if those wallets interact with fiat or reach certain volumes.

Solution	Custody Model	Integration Effort	Features & Capabilities	Compliance/ Security Posture
Web3Auth (SDK)	Non-custodial (user's device + distributed key shares)	Low – Few lines of code to integrate; includes UI widgets	*Passwordless social login:* Users sign up with Google, Apple, Facebook, etc., or via SMS/email OTP. Web3Auth then generates an Ethereum (or Solana, etc.) key pair where the private key is split between the user's device and Web3Auth's TEE nodes. *MPC key management:* No seed phrase; keys are reconstructed on-the-fly with user's login + Web3Auth network share [16]. Supports multi-factor auth and keys on multiple devices.	**Decentralized custody:** The provider never has full key alone – improves security and likely means the platform is not a legal custodian. Web3Auth uses secure enclaves and MPC for its share of the key, and the rest lives encrypted in the user's local storage or cloud backup under their control. Since users use familiar logins, there's less chance of lost keys. However, the platform should ensure **LGPD compliance** when using emails/social data and inform users how their data is used. No built-in KYC (anonymous to use), so if large transactions occur the platform might need to layer in its own compliance checks.

Thirdweb Wallet SDK	Non-custodial (user-controlled; can be device-stored or smart contract)	Low/Medium – Provides React, React Native SDKs and UI components; some setup of credentials	*Multiple wallet types:* Thirdweb supports **"Local wallets"** (private key stored on device, encrypted with password or biometric) [21], **social login wallets** (integrated with Web3Auth under the hood, or similar), and **smart contract wallets** (ERC-4337 accounts deployed per user) [17] [28]. *Comprehensive toolkit:* It offers an out-of-the-box "Connect Wallet" modal that can detect installed wallets and guide new users to create one in-app [29] [30]. Also includes features like transaction batching, gas fee optimizations, and support for 700+ EVM chains through its unified interface [18] [31].	**Open-source & modular:** Thirdweb's contracts and SDKs are open-source and audited. Smart wallets deployed via Thirdweb are fully on-chain and programmable, allowing policies like multisig, spending limits, or session keys. Security is high if best practices are followed (the smart wallet contracts are *"the most audited and battle-tested"* as they claim, and can require multiple approvals) [32] [33]. Since Thirdweb wallets are non-custodial, compliance is developer's responsibility – e.g. enforcing user approvals for transactions (which suits our HITL model) and adding any KYC gating externally if needed. Thirdweb doesn't perform user verification, but it makes wallet creation so easy (including "guest" ephemeral wallets) that the platform would need to ensure users are aware of tax obligations and perhaps restrict certain features until

Solution	Custody Model	Integration Effort	Features & Capabilities	Compliance/ Security Posture
				identity is verified, if required by law.

Table 1: Comparison of major wallet integration solutions by custody model, integration complexity, features, and compliance posture.

As seen above, using a **Wallet-as-a-Service API** (like Fireblocks or Coinbase Cloud) can jump-start the wallet functionality but requires more integration work and a commitment to managing compliance (KYC, reporting) on the platform's side. Meanwhile, **embedded non-custodial SDKs** (Web3Auth, Thirdweb) are very quick to integrate and user-friendly, and they offload much of the key management complexity to well-tested libraries – making them an attractive "plug and play" solution. For the Obsidian platform, a pragmatic path might be to start non-custodial (to avoid regulatory hurdles while validating the feature) and only move to a custodial or hybrid model if needed for additional features (like fiat on-ramps or interest-bearing accounts). Many platforms indeed start by simply letting users connect or create a self-custody wallet, then later possibly integrate a custodial offering for those who prefer convenience. We also can mix approaches: e.g. allow **connected external wallets (WalletConnect)** for crypto-native users *and* provide an **integrated Web3Auth/Thirdweb wallet** for users who want a one-stop, easy solution. This dual approach, as implemented by Unlock Labs for example, enhances accessibility. (Unlock integrated Coinbase WaaS to let users create wallets with Google login, while still allowing crypto-savvy users to connect MetaMask or WalletConnect if they have their own wallet [26] [34].) Providing both options covers compliance by segmenting users: those using the integrated wallet could be limited in functionality until they complete any necessary KYC, whereas those using external wallets presumably handle their own compliance (with the platform just facilitating access).

3. Integrating Crypto Balances and Transactions into the Dashboard

Once wallet functionality is in place, the platform will need to present crypto account data alongside traditional financial data in its dashboard. This involves fetching on-chain balances, transaction histories, and presenting them in a user-friendly, compliant manner.

Unified Account View: In Obsidian FinOps, users already have an account-based dashboard (likely showing bank balances, budgets, expenses, etc.). A crypto wallet can be introduced as another account/entity in this system. For each user (and each of their entities, e.g. personal vs business), the system can maintain one or multiple crypto addresses. These addresses and their holdings should appear in the net worth calculations, cashflow reports, and so on. The architecture supports multiple entities and already isolates records by Entity ID [35], which is useful: for example, a user could have a "Personal Crypto Wallet" under their personal entity and maybe a separate "Business Crypto" wallet under a business entity to keep transactions separate for tax purposes. The **multi-entity design** helps ensure that, say, personal crypto trades don't accidentally mix with business finances in reports [36] [37] – important for compliance with Brazilian tax rules (personal vs. CNPJ tax treatments).

Real-Time Balance Updates: To display crypto balances, the platform will query the blockchain (or a cached indexer) for each asset held. For a given Ethereum address, the relevant data includes the ETH balance and

token balances (ERC-20 tokens, etc.). Directly querying a blockchain node for each user and each token can be inefficient, so typically an indexer or API is used. As noted, services like **Covalent, Moralis, or QuickNode** provide single-call endpoints to get all token balances for a wallet. Covalent's "GoldRush" API, for instance, can *"fetch multichain ERC20, NFT and native token balances"* for a wallet across multiple chains [7]. Integrating such an API means when a user opens their dashboard, the backend can pull fresh balances (and possibly cache them for a short interval). Additionally, **price data** for crypto assets should be integrated so that the dashboard can show the fiat value (BRL) of the holdings. The platform might consume price feeds from an API (CoinGecko, CoinAPI, etc.) – e.g. Covalent can return historical price for a token at a specific date [38], which could help in portfolio performance charts or for end-of-year value (for IRPF reporting). The UI should likely show both the quantity of crypto (e.g. 0.5 BTC) and its current BRL equivalent.

Transaction History and Details: For each crypto wallet, the user will expect to see a list of transactions (similar to bank statements). This is more complex than balances because raw blockchain transactions are technical. An Ethereum transaction might involve internal smart contract calls, multiple token transfers, etc. The platform should leverage an API that provides *decoded* transaction history – for example, Covalent's transactions API returns *decoded event logs* and traces with human-readable info [39], and products like QuickNode's "Noves Translate" can convert a raw transaction into an English description (e.g. "Swapped 1 ETH for 2821 USDC") [40]. Using these services, the FinOps app could display crypto transactions in a friendly way (classifying them as "Sent to X", "Received from Y", "Swap trade on Uniswap" etc.). This aligns with the platform's focus on explainability and traceability. In fact, the architecture spec's concept of a *"Poker Table" change ledger* and audit trail [41] [42] can extend to blockchain events: every on-chain transaction should be recorded with metadata like timestamp, amount, counterpart address, and (if possible) a description of purpose (e.g. if initiated by an agent for a specific budget category). The platform can enrich on-chain data by allowing users or agents to tag and categorize transactions (e.g. marking a crypto transfer as a "business expense" or linking it to a project). Because Obsidian is an intelligent finance app, agents could assist by auto-categorizing crypto transactions using heuristics (like recognizing an address as a known exchange deposit, or detecting an NFT purchase vs a token swap, etc.), but always with user confirmation per HITL principles.

Multi-Chain and Token Support: A decision is whether to confine to one blockchain (perhaps Ethereum or a popular chain like Polygon) initially or support many. A FinOps platform might start with major assets (BTC, ETH, USDC, etc.). BTC is on its own network (UTXO-based) so integration would require Bitcoin-specific logic or an API (for instance, BitGo or BlockCypher for BTC). Ethereum and EVM chains are easier to manage uniformly. It might be pragmatic to begin with Ethereum chain support (which indirectly can allow holding tokens representing other things, like wBTC or stablecoins for real-world currency). Eventually, if users demand it, multiple chain wallets can be added (the UI might show each chain separately or abstract it away). Third-party APIs again make multi-chain easier by offering unified endpoints; for example, a single Covalent API key can fetch data from Polygon, BSC, etc., by just specifying the chain ID in queries [7].

Integration with Existing Ledger and Agents: The existing system uses a change-set ledger where all changes (transactions, categorizations, etc.) are events that can be approved [43] [44]. A crypto send transaction initiated either by the user or an AI agent should enter this same pipeline. For instance, if an agent proposes to *"transfer 0.1 ETH to savings"*, the system would log a pending change (with details like address, amount, rationale) and require user approval (HITL) before actually broadcasting the transaction. Once approved and broadcast, the transaction hash and status can be tracked and the final ledger updated when confirmed. Integrating with the ledger means even on-chain actions get an internal record ID,

enabling consistency with how other financial moves are tracked. The UI can then show a crypto transaction in pending state (awaiting confirmation on-chain or awaiting user approval) just as it does for, say, a pending bank payment initiated by an agent.

Compliance and Visibility: To assist with compliance, especially Brazil's tax rules, the platform should maintain **historical records** of crypto balances at year-end and a full log of transactions. For example, Brazil's Receita Federal requires that annual tax filings include the *quantity and value of crypto assets held on December 31* of each year (if above a modest threshold) [45]. The system can automatically snapshot user holdings on that date and generate a report. It should also track when a user sells or trades crypto, as capital gains may need to be calculated if monthly sales exceed certain limits (currently R$35k/month exemption threshold) [46] [47]. By having all transactions in a database, the platform can help users by computing gains and even pre-filling the Brazilian tax form (e.g. GCAP) details. This adds value beyond a normal wallet. Additionally, because Obsidian's design emphasizes auditability, the user could be given a **"crypto statement"** similar to a bank statement, listing all inflows/outflows, which can be useful for tax declarations or just personal records. Since all data is timestamped and immutable in the change ledger, the platform can defend the accuracy of these records if audited.

User Experience Considerations: In the dashboard UI, crypto accounts should probably have a distinct visual indicator (maybe a crypto wallet icon) to differentiate from bank accounts. When the user expands it, they might see sub-balances for each cryptocurrency held, or separate accounts for each major crypto. Showing the volatility is also key – perhaps charts of value over time, which the FinOps app can generate using historical price feeds. Agents could provide insights like *"Your crypto portfolio is up 10% this month"* or *"You spent $100 worth of ETH on fees this quarter"*, etc. These tie into the app's intelligence aspect. Finally, integrating crypto into budgets and cashflow: if the user receives crypto income, the system should allow categorizing it (e.g. an on-chain payment marked as income with proper tax classification, whether it's subject to IR or not). The architecture already contemplates classifying income/expenses and handling Brazilian tax categories (IRPF, MEI, ISS etc.) [37] – crypto could be another source of income or expense that needs classification (for example, if a user pays a contractor in crypto, that might need to be logged as a business expense with equivalent BRL value and maybe issue an NFS-e if required). Ensuring the system can convert crypto amounts to BRL at the transaction date's rate is important for proper accounting and tax compliance.

In short, integrating crypto data is about **pulling reliable on-chain data and fitting it into the FinOps context**. By using robust APIs to get balances and decoded transactions, the platform can display crypto holdings next to traditional finances with clarity. The design should maintain the same standards of traceability (every on-chain action recorded with who/why) and user oversight, so that the introduction of crypto doesn't compromise the "single audited path" principle [48] that the architecture upholds.

4. Secure Handling and Storage of Private Keys

Perhaps the most critical aspect of enabling a crypto wallet is how private keys are generated, stored, and used. Private keys are essentially the "keys to the kingdom" – if stolen or lost, assets are gone. Therefore, the platform must employ industry best practices for key management, tailored to whether it's custodial or non-custodial.

Client-Side Key Storage (Secure Enclaves & Key Stores): In a non-custodial scenario, the user's keys should reside on their own devices in secure storage. Mobile devices and modern OSes provide secure

enclaves/keystores – for example, **Apple's Secure Enclave/Keychain** on iOS and **Android Keystore** on Android allow apps to store cryptographic keys such that they are encrypted at the hardware level and not extractable [49]. When integrating a mobile wallet SDK, the platform should ensure it leverages these facilities (most SDKs like Web3Auth/Thirdweb do this by default for local keys, encrypting with the user's passcode or biometric). On web applications, secure storage is trickier (browsers don't have hardware enclaves accessible to web apps). However, Web Crypto APIs can store keys in IndexedDB and one could encrypt them with a key derived from a user password. Using a **Hardware Security Module (HSM)** or secure element on the client side is basically what hardware wallets (like Ledger or Trezor) do – they keep keys in a separate hardware chip. The platform could allow optional integration with such hardware wallets for advanced users, adding another security layer.

Server-Side Key Management (HSMs & MPC): If the platform or its partner is holding keys on servers (custodial mode or partially custodial), then dedicated HSMs or MPC setups are mandatory. An **HSM** is a specialized device (could be a network appliance or cloud-based like AWS CloudHSM or Azure Key Vault HSM) that stores keys and can sign transactions inside a tamper-resistant environment. Even administrators of the system cannot extract the keys from an HSM; at most they can instruct it to use the key for signing if authorized. HSMs often come with FIPS 140-2 Level 3+ certification (high security). The platform could also use cloud-based enclaves (e.g. AWS Nitro Enclaves) to isolate the signing process. Another modern approach is **Multi-Party Computation (MPC)**, where the private key is never assembled in one place – instead, multiple servers each hold a share of the key and collaborate (via cryptographic protocol) to produce a signature. Fireblocks and Coinbase WaaS both use MPC. The benefit is no single machine has the full key, and it also enables geographically distributed signing (and key recovery if one node fails). With MPC, one can also distribute key control between the platform and the user or between multiple servers the platform controls. For example, the platform could keep one key share in its database (encrypted) and another share could be derived from the user's login credentials – both needed to sign. This way, even if the database is compromised, the attacker cannot sign transactions without the user's secret. **Threshold signatures** (a form of MPC) can be set such that any M of N shares can sign; this can be leveraged for redundancy or requiring multiple approvals (akin to multisig but at key level).

Backup and Recovery: A key management strategy must include how to handle backups and user key recovery. In a self-custody model, typically users are given a **seed phrase** (12-24 words) to write down as a backup. However, expecting mainstream users to safely store seed phrases is risky (they might lose it or not secure it properly). Alternatives include: storing an encrypted backup of the key in the cloud (protected by the user's password or device lock). For instance, some wallets encrypt the seed with the user's iCloud/Google Drive credentials (or a user-provided passphrase) and back it up – so if they lose the device, they can recover via cloud. Social recovery is another option: EIP-4337 smart wallets support "guardians" who can collectively restore a wallet if the user loses access. The platform could allow the user to designate a couple of friends (or their own devices) as guardians. If the user loses their key, those guardians can approve a reset after a timelock. This is complex to implement but increasingly available in account abstraction frameworks [50] [51]. For custodial wallets, backup means redundancy on the provider side (like backing up key shares in multiple data centers, and having an offline cold backup). From the user perspective on a custodial wallet, recovery is simply contacting support (since the provider holds the keys, the user "recovering account" is more about identity verification to ensure the right person gets access again).

Secure Transaction Signing Process: When a transaction is to be signed, the process should ensure the key is used in as isolated and user-authorized a way as possible. In a user-controlled mobile wallet, this means requiring biometric or PIN confirmation to unlock the key (leveraging the device's secure enclave to

do the signing so the key isn't exposed in memory). In a custodial service, transaction signing requests should go through an approval flow and the signing should occur inside an HSM/MPC node. Fireblocks, for example, uses an **M of N signing scheme** where certain operations might require quorum approval (multiple admin users must confirm in the console), especially for large transfers – the platform could configure such policies so that high-value transfers require a second factor or manual admin approval (like a "four-eyes" principle).

Preventing Key Exposure in the Application: Developers must be careful that no sensitive key material ever gets logged or transmitted insecurely. The code should avoid printing keys to logs, and API calls that handle keys (if any) must use encrypted channels. If the platform uses a browser environment for some reason (e.g. a web app wallet), it should consider using a browser extension context or native wrapper to store keys, rather than regular web local storage which is vulnerable.

Utilizing Policy-as-Code for Key Usage: The Obsidian platform has a notion of **policy-as-code** for approvals and changes. This can extend to how keys are used. For instance, one could encode policies that "private key X cannot send more than Y amount per day" or "transactions to new addresses require an extra confirmation". Some of these can be enforced off-chain by the platform's transaction guardrails, but interestingly, if using a smart contract wallet, they can also be enforced on-chain (the smart wallet contract could have spending limits or require multiple signatures above thresholds). As a simpler measure, the platform could maintain an internal policy file that the signing service checks: e.g., if an agent tries to sign a transaction not approved by the user, the signing module refuses – effectively the signing function consults the change-set ledger or approval status before proceeding. This ties into security: even if an attacker somehow triggered the signing function, it wouldn't sign because the policy condition (user_approved == true) isn't met.

Hardware Isolation: For maximal security (especially if high amounts are involved), the platform should consider keeping the majority of funds in **cold storage**. Cold storage means keys that are completely offline (e.g. on a hardware wallet or an air-gapped machine) – these funds are not accessible for immediate spending but can be moved to hot wallets when needed. In the context of a FinOps personal wallet, this might not be applicable if every user has their own wallet (each user would manage their own "cold" storage amounts). But if the platform aggregated funds or had treasury pools, it would certainly use cold storage for long-term holding. On a user basis, the platform could encourage users to use hardware wallets for large holdings by integrating with Ledger Live or similar (that might be beyond initial scope, though).

Logging and Monitoring Key Usage: Every usage of a private key (especially on the server side) should be logged with time, what was signed, and by which process – this complements the platform's audit philosophy. If an admin triggers a key export (which ideally should be either impossible or heavily controlled), it should be logged and alerted. In Fireblocks' system, they have the concept of transaction and policy logs for all admin actions [52]. The Obsidian platform can similarly log agent or user actions that led to a signing event, linking it to the resulting transaction hash for complete traceability.

User-Controlled vs Platform-Controlled Keys: It's worth explicitly deciding if this will be a **user-controlled wallet** (non-custodial) or a **platform wallet** for the user. In user-controlled cases, private keys (or key shares) should only ever be in the user's possession (their device). The platform might assist in setup but after that, the user's key doesn't leave their device. In platform-controlled (custodial), the platform needs to secure it akin to how banks secure passwords/funds. A compromise of keys is akin to a bank heist. Thus, if platform goes custodial, heavy investment in security audits, infrastructure and possibly insurance is

recommended. Even when using a third-party custodian, the platform should conduct due diligence on their security (Are they SOC2 certified? Have they had breaches? Do they have a bug bounty? etc.).

In summary, for non-custodial integration, rely on **device OS security features** and well-tested libraries (don't roll your own crypto). For custodial or semi-custodial, use **HSM/MPC technology** and enforce dual controls on key usage. Always have a **secure recovery method** for users, because forgetting a seed phrase is common – solutions like MPC splitting (as Coinbase WaaS does to allow key recovery) [15] or social recovery can avoid catastrophic loss. And from a process standpoint, integrate key management into the platform's overall security model: for example, if the platform has a master "kill-switch" to freeze all operations (as mentioned in architecture for emergency) – extend that to crypto by, say, flipping all wallets into a "locked" mode where no transactions will sign until unlocked by admin. This could be done via a flag that the signing module checks (and even on-chain if smart contract wallets by having an emergency pause function). The combination of these measures will ensure private keys – the heart of the crypto wallet – are handled with the utmost care, matching or exceeding the security of traditional financial credentials.

5. Smart Contract Wallets and Account Abstraction (EIP-4337) Integration

Smart contract wallets represent an evolution in wallet technology that could align well with Obsidian's agent-based, policy-driven architecture. Unlike an Externally Owned Account (EOA) which is controlled by a single private key, a **smart contract wallet** is a programmable contract on the blockchain that holds funds and defines its own rules for transaction authorization. Ethereum's **EIP-4337 (Account Abstraction)** standard has made it possible to use smart contract wallets with features like meta-transactions and batched operations without modifying the core blockchain protocol [53] [54]. We explore how such wallets work and how they could fit into the existing "agent proposes, approval flow, then commit" model.

What is a Smart Contract Wallet? It's essentially a custom logic account. For example, the **Safe (formerly Gnosis Safe)** is a popular smart wallet where the logic is "require M-of-N signatures for any outgoing transfer" [33]. Other logic could be "allow a daily withdrawal limit" or "enable social recovery by these guardians". Traditional EOAs can't do this – they're all or nothing with a private key. Smart wallets, by being contracts, can incorporate **multi-signature**, **whitelisting**, **limits,** and interact directly with complex DeFi protocols through internal logic. This aligns with Obsidian's idea of policies and approvals. In effect, the on-chain wallet could enforce the same constraints that the off-chain FinOps system wants to guarantee. For instance, we could deploy a wallet contract for the user that requires any transaction to be approved by the user *and* flagged as approved by the off-chain orchestrator (this could be done by having the contract require two signatures: one by the user's key and one by a platform key that signs only if the off-chain conditions are met).

EIP-4337 Account Abstraction: ERC-4337 (recently live on Ethereum) enables a new transaction flow: instead of EOAs sending transactions, **User Operations (UserOps)** are posted to a separate mempool, and **Bundlers** pick them up and feed them into an **EntryPoint** contract on-chain which then calls the target smart wallets [55] [56]. In practice, this means a user's smart wallet can initiate transactions (via a UserOp) that a third-party bundler pays gas for, enabling **gas abstractions** (paying fees in ERC-20 or via a sponsor) [57] [58]. It also means multiple actions can be bundled – e.g. the wallet might do three calls in one operation (batching, which is great for user experience like bundling an approval + a swap + a transfer) [59]. And crucially for security, ERC-4337 wallets can implement things like **multiple signatures** and **session**

keys directly. For example, a wallet can require two signatures (maybe one by the user and one by a guardian or a policy module) for certain ops, or allow an agent to spend up to $X with just its signature and beyond that require the user's as well – these kinds of "transaction rules" can be coded.

From the perspective of Obsidian FinOps' agent ecosystem, one could imagine each AI agent might even have its own smart contract wallet (or a sub-wallet with limited permissions). The audit report mentioned *"agents as independent economic actors with their own wallet and budget"* [60]. Currently that was conceptual (likely just an internal ledger per agent), but this could be realized on-chain: each agent could be a designated signer on a user's smart wallet with a spending cap. For instance, the budget limits enforced off-chain could be mirrored by giving each agent's key the ability to send a UserOp from the wallet contract with a built-in limit (the contract could check: if this agent-key initiates, limit amount). More straightforwardly, the **user's smart contract wallet** can be integrated with the approval flow by requiring an off-chain approval signature for every operation.

Example – EIP-4337 Wallet with Off-chain Approval: We could design the wallet such that it only executes a UserOperation if it includes a signature from a special "guardian" key held by the FinOps platform's server which signifies "this operation was approved via the change-set process". The flow might be: an agent drafts a transaction (say, transfer 0.5 ETH to Bob's address). This goes into the change-set ledger with agent's rationale. The user approves it in the FinOps UI (HITL). Upon approval, the platform's server assembles a UserOp (the ERC-4337 transaction object) and signs it with the platform's guardian key and perhaps partially with the user's key if the user's device cooperates. Then it sends it to a bundler to execute on-chain. The smart wallet contract, when receiving this operation, will verify that the platform's guardian signature is present. If some attacker (or rogue agent) tried to send an operation without that signature, the contract would reject it. This effectively extends the off-chain policy enforcement onto the blockchain level, providing defense in depth. It's analogous to how Safe requires multiple people to sign – here one of the "people" is the off-chain policy mechanism.

Smart Contract Modules and Extensions: Smart wallets like Thirdweb's or Safe can be extended with modules. For instance, Safe has modules for daily spending limits, or to allow an authorized module contract to execute certain transactions without full approval (often used for things like gasless transaction relays for small amounts). In our context, we might use a module that allows an agent to perform micro-transactions (say under a certain BRL value) without bothering the user each time, if the user opts in to that autonomy gradually (as the spec envisioned *"progressive autonomy"* where low-risk actions might auto-approve eventually) [61]. A "LimitModule" could be added to the wallet that says transactions under ₹X (or equivalent) to known addresses can be executed with just the agent's signature, otherwise require full confirmation. These kinds of on-chain policy modules align with the *"policy-as-code"* ethos – except here policy-as-code literally lives in the smart contract (auditable and transparent).

Safety and Compliance with Smart Wallets: Smart contract wallets do have overhead (gas costs are a bit higher than simple EOAs) and complexity (smart contracts can have bugs). Fortunately, standards like Safe are extremely battle-tested [62] [32] and new AA wallets being deployed often use templates that are audited. Still, the platform should rely on known implementations (e.g. Thirdweb's base contracts or Safe's contracts) rather than writing one from scratch. Another advantage for compliance: since smart wallets are contracts, they can emit events that are easier to track for auditing. For example, a custom event could log the agent ID and policy ID for each transaction executed. Tools could then pick that up and integrate it into the off-chain ledger (essentially linking the on-chain TX to the off-chain proposal ID via an event).

EIP-4337 and Meta-transactions: With account abstraction, the gas for transactions could be sponsored by the platform (via a **Paymaster** contract). This might be useful to improve UX: new users might not have ETH for gas, so the platform could cover the first few transactions' fees (and maybe charge it in fiat or as a subscription cost). EIP-4337 allows gas fees to be paid in alternative tokens or by an external sponsor [58]. We could integrate a Paymaster that accepts, say, DAI or USDC (or even BRL stablecoin) from the user's balance to pay fees, making the experience more seamless (user doesn't have to manage ETH just for gas). This must be done carefully, as sponsoring unlimited gas could be abused – likely only small routine transactions or initial setup would be sponsored.

Integration with Agents: The architecture's notion of *"Change Manager service"* validating proposals before committing [48] plays well with smart wallets because the actual blockchain commit could be the final "commit" of a change. The **bundler** that sends the UserOp could be run by the platform or a reliable third-party (StackUp, Infura, etc., run public bundlers). The platform might run its own bundler to have more control and ensure timely inclusion of transactions, but it's not strictly necessary. Agents that are proposing EIP-4337 transactions would essentially be preparing UserOperation objects (with target, data, etc.) instead of normal transactions, but the conceptual difference is minor from the platform's perspective.

In summary, smart contract wallets can bring the platform's security model on-chain: multi-sig approvals, time-locks, spend limits, and even the *"kill-switch"* can be implemented in the contract (e.g. an emergency pause that the user or platform can trigger to temporarily block all outgoing transfers if suspicious activity is detected). Given that Obsidian FinOps is about being a "100% compliant" financial OS, having an on-chain wallet that is *programmable to enforce compliance and approval logic* is a powerful complement. It ensures that even if an agent or other component misbehaves, the blockchain won't execute unauthorized transactions.

Concretely, the platform could start with a simpler approach (like a multi-sig Safe wallet where the user's key and an "approval key" held by the service are both required). Over time, as EIP-4337 infrastructure matures, it can migrate to a more flexible AA wallet that doesn't require user to have ETH for gas and can incorporate more complex logic directly. This is forward-looking, but since the question explicitly mentions *"agents propose EIP-4337 transactions"*, it implies planning for account abstraction. Real-world example: **StackUp's** bundler API or Thirdweb's smart wallet SDK can handle much of the heavy lifting, allowing the platform to issue UserOps after internal approval and have them mined via a bundler network.

To visualize, one of Thirdweb's diagrams shows how different wallet types compare – custodial vs MPC vs smart wallets – and highlights that smart wallets enable **"programmable security"** (on-chain rules) [63]. This aligns exactly with giving the platform's policy engine a handle on the wallet. By using smart contract wallets, the platform essentially extends its "Compliant OS" philosophy to the blockchain itself, embedding compliance into the very transactions (truly turning "code is law" to our advantage by encoding business law and rules into the wallet contracts).

Diagram: Example of a smart contract wallet enabling programmable transaction rules (e.g. requiring an approval signature or multiple signers) that can be upgraded over time. Such wallets allow on-chain enforcement of policies, aligning with the platform's change-set approval model. [32] [33]

6. UI/UX Requirements and User Flows for Crypto Onboarding & Transactions

Introducing crypto functionality to a FinOps platform means designing user experiences that cater both to crypto-savvy users and newcomers. The UI/UX must make tasks like wallet setup, sending/receiving assets, and monitoring balances as simple and reassuring as possible, while also educating users about the differences from traditional finance.

Onboarding – Wallet Creation: The first step is how a user gets a crypto wallet in the platform. Ideally, this should be seamlessly integrated into the existing onboarding or profile setup. If using a Web3Auth or Coinbase WaaS approach, the user could be presented with an option like **"Enable Crypto Wallet"** and then allowed to log in with Google or create a password for wallet backup. For example, Unlock Labs' integration of Coinbase WaaS completely revamped their wallet creation flow to make it *"as straightforward and secure as possible"* for both new and experienced users [64]. They introduced multiple sign-in options including Google OAuth to let users create wallets with an existing account [26]. We should follow a similar pattern: allow creation via familiar methods (social login, email link, etc.) for novices, and also allow those who already have a wallet to simply connect it. The UI might ask: *"Do you have a crypto wallet you'd like to use?"* – if yes, they can connect via QR code or browser extension; if not, the app will create one for them in a couple of clicks. During wallet setup, educate the user about security: if a seed phrase is provided, walk them through writing it down (perhaps even have them confirm a couple of words to ensure they saved it). Many modern wallets try to avoid showing the seed phrase immediately to not scare users; they instead use cloud backup or postpone the backup step until the user has some funds. We might adopt a similar strategy: create the wallet without immediately forcing the user through a complex backup step, but remind them to back up later (with persistent warnings until done).

Receiving Crypto (Deposits): The UI should allow the user to receive funds, which typically means showing their wallet address (and QR code) for each asset or chain. For simplicity, if using an EVM wallet, one address can receive ETH and all ERC-20 tokens, etc. We'd show a QR code that encodes the address (perhaps in both plain hex and as a QR for scanning). We should include warnings like *"Send only compatible assets to this address. Sending any other assets may result in loss."* (E.g. if it's an ETH address, don't send Bitcoin to it). For user friendliness, the app could detect if the user has copied an address from somewhere (some wallet apps do this) to offer quick actions like "You copied an address, do you want to add it as a contact or verify it?". An interesting FinOps twist: if the user is expecting a payment (say a client paying in crypto), an agent could generate a deposit address and possibly create an "invoice" object linking the expected payment, then detect when funds arrive. But MVP can be manual: just display address and maybe allow the user to share it via email or WhatsApp directly from the app (with a message like "Here is my payment address").

Sending Crypto (Payments/Transfers): The send flow should feel as close to sending a normal payment as possible, but with crypto-specific fields. The user will need to enter a recipient address – this is a long hex string, which is error-prone to handle. UX measures: incorporate a **"Paste from clipboard"** button (since many will copy an address from somewhere), and validate the address format immediately (e.g. checksum for ETH addresses) to catch typos. If the platform supports ENS (Ethereum Name Service) or other naming, allow entering a human-readable name (like alice.eth) and resolve it. Provide an address book feature: after a user sends to an address, let them save a nickname so that future transfers show "Bob's Wallet" instead of raw 0xabc123. The send form will include the amount – allow switching between entering in crypto (e.g. 0.05 ETH) or in fiat (e.g. BRL 500, which the app converts to crypto amount using current rate). This is

important because many users think in fiat terms. Next, the UI should display any fees. For a given transaction, estimate the network fee and show it in both crypto and fiat (e.g. "Network fee ~0.001 ETH (~R$8)"). If the wallet allows fee customization (slow vs fast), maybe have a simple option for "Economy / Regular / Priority" speeds with corresponding fee estimates. However, too many options might overwhelm – an intelligent agent could automatically choose a reasonable fee. The user then confirms and, depending on custody mode, either is prompted to sign the transaction or just to confirm they want to proceed and the platform handles signing. If it's a user-held key (non-custodial), a prompt from the secure element or external wallet will appear – e.g. FaceID to confirm on iPhone or a MetaMask popup on web. If it's custodial/MPC, the platform might just require 2FA or a PIN to confirm, then do the signing on server side. This step is crucial to **prevent accidental or malicious transactions** – always require an explicit user action (except maybe for automated micro-transactions under user-defined rules).

Transaction Approval Flow (HITL): For agent-initiated transactions, the UI needs to present the proposal to the user in a clear way. For example, *"AI Budgeting Agent proposes to swap 100 USDC for 0.05 BTC for long-term savings."* Show details: current exchange rate, any fees, rationale if provided. Then buttons: Approve or Reject. This should integrate seamlessly with the existing approval inbox that user likely has for other changes (as per the spec: "almost nothing is auto-approved at first" [65]). Once approved, the status changes to processing, and then to done (or failed if something on-chain failed). If the user rejects, ideally allow sending feedback to the agent (to learn/improve).

Portfolio and Analytics: The crypto wallet section of the app should not just show raw transactions, but also analytics in the spirit of FinOps. For example, show **asset allocation**: X% in BTC, Y% in ETH, etc., maybe with a pie chart. Show **performance over time**: since crypto is volatile, a timeline chart showing total wallet value over the past day/week/month is useful. If the platform already has budgeting/forecast modules, integrate crypto there as well – e.g., allow users to set a budget for how much fiat they convert to crypto each month (and track it), or an agent might suggest *"Convert your unused travel budget to a stablecoin savings"*. On the flip side, if user spends crypto, incorporate that into expense tracking (with category if possible).

Education and Safety UX: Many users are unfamiliar with crypto specifics like the irreversibility of transactions and need for self-protection. The app should include just-in-time education. For example, when the user is about to send to a new address, maybe show a tip: *"Always double-check the recipient address; crypto transactions can't be reversed!"*. Or if a user copies an address to clipboard, maybe have the app detect if a known scam address was pasted (some apps have a database of scam addresses). Since Brazilian users have to consider tax, maybe when they do their first crypto sale, prompt: *"Don't forget: crypto sales may be taxable if over R$35k/month – we'll help you track this."* basically prepping them.

LGPD & Consent UX: Because crypto might involve personal data usage (like if the app integrates an exchange requiring CPF, or if it collects addresses which might be personal data if tied to identity), ensure there are clear consent screens. For instance, *"To activate your crypto wallet, we need to collect some information and share with our custody partner for compliance."* – with an accept step. Also a disclaimer: *"Cryptocurrencies are volatile and not insured like bank deposits"* could be presented in onboarding to manage user expectations and legal requirements.

Localized UX for Brazil: Brazilian users are quite used to easy fintech experiences (PIX instant payments, for example, has set a high bar). The crypto wallet UX should try to mimic the ease of PIX where possible (PIX is instant and uses simple keys like phone/email; crypto is more complex, but perhaps using ENS or

having user's contacts with crypto addresses stored can approach that ease). Additionally, consider language: use terms that Brazilian users are familiar with (e.g., many might refer to crypto trading terms in English, but whenever possible provide Portuguese clarifications in tooltips or the like if the app is localized). However, the user asked for results in English, so presumably the app might be English for now.

Wireframes & Flow Diagram: Conceptually, an **onboarding flow** might look like: 1. User navigates to "Crypto" section for first time → sees a welcome explaining the feature and a button to "Set up my Crypto Wallet". 2. Click → If not KYCed, ask for identity verification (maybe integrate with onfido or govt ID if custodial). Or if non-custodial via social, ask them to choose login method. 3. After setup, show a screen "Your Wallet is Ready" with the address and perhaps a one-tap button to fund it (could integrate a fiat on-ramp provider to buy crypto with a card, which is another integration to consider). 4. Subsequent visits → goes straight to wallet dashboard (balance, etc.).

For **sending flow**: 1. Go to Send, enter recipient or choose from contacts, enter amount. 2. If the user chooses a contact that has an email but no crypto address on file, maybe the app can generate a payment link? (Some wallets do this: they create a link that the recipient can click to provide an address or even create a wallet). 3. Confirm details → confirm with security (fingerprint/PIN). 4. Show "Transaction pending..." then "Sent" with transaction hash and an option to view on block explorer.

For **receiving flow**: 1. Tap Receive, select which asset (if multiple) → display QR and address. 2. Perhaps allow user to share it via WhatsApp/Telegram directly from app.

Human-in-the-Loop UX: Because every significant action is meant to get user approval [65], the UI should have a clear **notifications/approval** center. If an agent schedules a transaction or one is awaiting approval, it should be visible (possibly as a badge on a "Tasks" or "Approvals" icon). The user can review details and approve. This is a distinguishing factor: unlike normal wallets where a transaction either is user-initiated or not at all, here transactions could be agent-initiated. So we have states like *"Proposed by Agent, waiting for your approval"*, which the UI should represent clearly (maybe in the transaction list with a special icon or section). After approval, maybe *"Executing..."* until confirmed on-chain, then *"Completed"*. If rejected, show as *"Rejected by you"* so the agent knows it (and maybe agent can take alternative actions).

Error Handling: If a transaction fails (maybe gas too low or contract error), the user should be alerted and given guidance. Perhaps an agent can automatically catch a failure and suggest a remedy (e.g. *"Transaction didn't go through due to low gas. Shall I resubmit with a higher fee?"*). This kind of resilience will help user trust.

Incorporating Compliance in UX: When needed, bring compliance steps into the UX gracefully. For example, if user hits a threshold that legally requires info (like making >R$30k of transfers which triggers RFB reporting), the app could pop up a note: *"We noticed high transaction volume this month. Brazilian regulations require reporting such transactions. We will generate a report for you."* and maybe provide a one-click way to get that monthly report or guidance to submit to tax authorities. Similarly, for LGPD, provide an easy way to get a copy of their data or delete their wallet if they choose (which in crypto means maybe transferring out funds and deleting keys).

Accessibility and Simplicity: Keep UI elements simple, avoid crypto jargon where possible. Instead of "Sign Message" say "Confirm Action". Instead of "Gas Price", say "Network Fee". Provide info icons for more advanced details for those who care. Given FinOps is likely targeting not hardcore crypto users but everyday

people handling finances, lean on analogies: e.g. call the wallet a “Crypto Account” in some places, because people understand bank account. But within help tips, explain it’s a blockchain wallet.

Consistency with FinOps Style: The wallet UI should adopt the same design language as the rest of the app. If the app has a graph-based “Chip view” for finances [66], maybe crypto accounts show up as chips (e.g. a node for “Crypto Wallet” linking to transactions as edges). Or if there’s a grid dashboard, include a widget tile for “Crypto Portfolio”. The user should not feel they left the app – it should feel like just another part of their financial picture.

In conclusion, the UX should strive to **demystify crypto** for users: guide them through wallet setup in a few easy steps, ensure sending/receiving is as close to “Venmo” or PIX simplicity as possible (with added precautions for addresses), and integrate the approval workflow so users remain comfortably in control. By prioritizing clarity (showing where money is going and why), confirming intent at key steps, and giving continuous feedback (status of transactions, etc.), the platform can make crypto a natural extension of its financial OS capabilities.

[26] [34] illustrate how multiple sign-in options (Google OAuth, existing wallets via WalletConnect) can be offered to streamline onboarding for different user preferences. Following such real-world examples will help ensure our UX meets users where they are – whether crypto native or completely new.

7. Brazilian Legal and Tax Compliance Considerations (IRPF, Receita Federal, LGPD)

Integrating crypto into a Brazilian financial app requires careful attention to local laws on taxes, reporting, and data protection. Brazil has been proactive in crypto regulation, so the platform must incorporate features to keep users compliant with minimal effort and abide by the legal requirements itself.

Tax Reporting (IRPF and Monthly Declarations): Brazilian taxpayers are required to declare their crypto holdings and certain transactions to the Federal Revenue Service (Receita Federal). Key points: - In the **annual Income Tax (IRPF) return**, any cryptocurrency assets above BRL 5,000 in value must be declared as assets (even if not sold) [45]. This means if a user holds more than R$5k worth of crypto on December 31, they must list it in their tax return, including details like type of asset, quantity, acquisition date and cost. The platform can assist by providing a summary at year-end: e.g. *“On 31/12/2025, you held 0.1 BTC valued at R$X and 2 ETH valued at R$Y.”* The Tax Intelligence module already planned in the architecture could handle multi-entity separation so that personal vs business holdings are properly segregated for reporting [36] [37]. - For **capital gains tax**, Brazil exempts gains on crypto sales up to a certain limit (currently sales up to R$35,000 in a month are exempt from capital gains tax) [46]. Above that, gains are subject to tax (15% on gains usually). The platform should track when a user sells or trades crypto for fiat or another crypto and calculate the gains. If a user sells more than R$35k in a month (or the law’s current threshold), they owe tax on the profit of those sales. This calculation can be complex (need to consider cost basis of assets sold). The platform could integrate with tax calculation libraries or services (like TokenTax or CoinTracker) to automate this. At minimum, it should alert the user: *“You sold R$50,000 worth of crypto in August. Approximate capital gain is R$10,000 – this may need to be reported and taxed.”* The architecture’s emphasis on categorizing income/expenses can extend to categorizing crypto gains as taxable income. - **Monthly reporting (IN 1888/2019):** Brazil’s Normative Instruction 1888 requires that Brazilian residents **report their crypto transactions to Receita Federal on a monthly basis if they exceed R$30,000 in a month** (and if these

transactions aren't already reported by an exchange). Specifically, if a user uses foreign exchanges or self-custody wallets and in one month the total value of transactions (buys, sells, transfers) is above 30k BRL, they must submit a report by the end of the following month detailing those transactions [67]. This is a separate obligation from the annual tax return. Our platform can hugely help here: by monitoring user transactions, it can know if that threshold is crossed. If so, the platform could generate the required report (in the format Receita expects, possibly a CSV or an online form submission). At the very least, notify users: *"You transacted R$45,000 in crypto this month. Receita Federal requires a monthly report (IN1888). Click here to download a report of your transactions or see instructions."* We might even consider an integration to automatically submit this if user provides e-CPF credentials, but that might be beyond scope initially. Ensuring users don't inadvertently become non-compliant is a big value-add, since penalties for failing to report are fines (ranging R$100 to R$500) [68].

By implementing these, the platform positions itself as a **compliance-friendly** solution, not just a tech tool. This is in line with being a "100% compliant OS".

LGPD (Lei Geral de Proteção de Dados): As mentioned in section 1, our platform must handle personal data carefully. Crypto wallet integration can involve additional personal data in a few ways: - **KYC data:** If we perform KYC for a custodial wallet, we will collect documents, selfies, etc. These are sensitive personal data. LGPD mandates explicit consent for processing such data and outlines rights like access, correction, and deletion. The platform must securely store this information (likely encrypted, separate from other data) and only use it for compliance purposes. Users should be able to request deletion of their personal data (though if it's a regulated necessity to keep for X years, we might anonymize or securely archive it rather than full deletion). - **Blockchain addresses:** Interestingly, whether a blockchain address is considered personal data under LGPD could be debated (it's pseudonymous – not directly identifying, but could be linked to a person). However, if our platform links an address to a user account, then that linkage is personal data we control. We should treat it as such, meaning not exposing addresses unnecessarily and protecting that mapping. - The architecture already planned for LGPD compliance: *"requiring user consent for data access, providing data deletion, and strong audit logs and RBAC"* [12]. Adding crypto doesn't change those fundamentals but we should update our privacy policy to mention what crypto data we collect and why (e.g. transaction history to provide the service and for compliance reporting).

Regulatory Status and Licensing in Brazil: As of 2024, Brazil is in transition to a new regulatory regime for crypto. Law 14.478 (passed in 2022) set definitions and empowered the Central Bank to regulate. A decree in 2023 clarified roles. By 2024-2025, the Central Bank is expected to start issuing licenses for VASPs. Likely categories include brokerage/exchanges and custody providers. If our platform takes custody of crypto, we will likely need to either obtain a license or partner with someone who has. The timeline (per Notabene's summary) shows public consultations and phased implementation [69] [70]. Possibly by 2025 the rules for custody license will be in effect. To be safe, the business should engage with regulators or legal counsel in Brazil early. In the meantime, since crypto is fully legal in Brazil and widely used, operating with best practices (KYC, AML controls, cooperating with tax authorities) will put us in a good position for licensing. If non-custodial, we might avoid needing a license because we're not "holding or managing assets for others" (the legal nodes guide noted non-custodial providers usually don't need authorization) [71] [3]. But even non-custodial services could be encouraged to register with regulators just to be on record. We should stay updated on BCB's regulations and possibly register our service if required.

Consumer Protection and Other Laws: Brazil has strong consumer protection laws (and a very active Procon). The platform should be transparent about risks – e.g. include a disclaimer that crypto investments

are not guaranteed, possibly also disclaim liability for price volatility or on-chain errors outside our control. Also, the user agreement should clarify that by using the crypto service, the user agrees to our handling of their data for compliance (LGPD requires specifying purpose). Since the FinOps platform also deals with personal and business finance, integration with crypto means we should ensure any cross-border aspects (if using foreign APIs or custodians) are okay under Brazilian law (for instance, if user data is stored on U.S. servers, comply with international transfer rules of LGPD, like using standard clauses or the user's consent).

Integration of Tax Module: The audit mentioned a dedicated "Tax Intelligence" module to handle Brazilian tax rules (e.g. IRPF categories, ISS for services) [72]. That module should be updated to handle crypto specifics. For instance, classify crypto trades under capital gains and track them for annual reporting. Or if the user is an MEI (micro-entrepreneur) paying suppliers in crypto, note that while the payment is in crypto, it might still require a Nota Fiscal issuance for the service. If the platform can't directly issue NFs for crypto payments yet, it can at least prompt the user to issue one externally.

Reporting to Authorities: Under IN 1888, exchanges have to report user transactions monthly. A self-custody wallet provider may not have an obligation to directly report (since that falls on individuals), but if we become custodial, we might have to report all user transactions above certain thresholds. It's not entirely clear if the new law will force custodians to do similar reporting to the Central Bank. Likely yes, as part of prudential supervision. We should implement the ability to generate required compliance reports (like a ledger of all crypto withdrawals/deposits by users, with CPF info attached) in case regulators request it. Having an immutable change-set ledger internally helps ensure we have accurate records to draw from [42].

Anti-Fraud and AML: Brazil's regulations will require AML programs. That means monitoring transactions for red flags like structuring (breaking transactions into smaller ones to avoid reporting), unusual patterns, or known illicit addresses. We may integrate a blockchain analytics tool (like Chainalysis, Elliptic, or Scorechain). In fact, QuickNode marketplace shows "Scorechain Risk Assessment API" for AML/CFT analytics [52]. We could use such a service to score incoming funds (e.g. if a user receives crypto from a wallet flagged as dark market, we could flag or freeze it pending review). This protects both us and the user (since unknowingly transacting with tainted funds could cause legal trouble). Any such action should be done in line with terms of use and Brazilian law (for example, if we freeze funds, is there legal basis? Typically yes if it's suspected criminal funds, but we'd involve authorities).

LGPD specifics: We should ensure to get user consent for any new data usage not covered by existing consent. For example, if we decide to share some user data with a third-party custodian (Coinbase or Fireblocks), we must disclose that in our privacy notice and maybe even in a consent checkbox, since LGPD requires informing about international data transfers (if applicable). Also, allowing the user to delete their crypto account – tricky if it's non-custodial (they have the keys, we just would forget the linkage). But if custodial, a user may request account closure; we'd then assist them to withdraw funds and then close out their account (retaining records only as long as legally required).

In conclusion, Brazil's environment means our platform should be as much a **tax assistant and compliance tool** as it is a wallet. We'll build in features to: - automatically **generate tax reports** for annual filings (with asset values and any realized gains) [45], - **alert and assist** with the RFB monthly transaction reports when thresholds are exceeded [67], - maintain strict **data protection** practices (consent, secure storage per LGPD) [12], - and prepare for upcoming **licensing** by aligning with best practices now (KYC, AML monitoring, audit logs of all transactions with user identification).

The platform already has a solid compliance foundation (multi-entity data isolation for personal vs business, audit trails, tax classification) [36] [37]. Extending that to crypto means treating crypto transactions like any other financial transaction: fully logged, categorizable, and reportable. If done right, the platform can advertise itself as a **compliant gateway** to crypto – a differentiator in a landscape where many crypto apps leave compliance entirely to users. This could attract users who want the benefits of crypto but are worried about running afoul of rules – our app will have their back, automatically.

8. Trade-offs of Different Architectural Choices (Embedded vs. WaaS vs. Full Node Integration)

When integrating crypto wallets, the system architecture can range from fully relying on external services to building everything in-house. Each approach has trade-offs in terms of control, complexity, security, user experience, and scalability. Here we compare a few key choices:

Embedded Non-Custodial Wallet (self-managed) vs. Custodial Wallet-as-a-Service:

- *Development Effort:* Using an **embedded wallet SDK or self-custody approach** (like integrating Thirdweb or Web3Auth) is generally faster to develop. The heavy lifting of key management, transaction signing, etc., is handled by the library. In contrast, adopting a **custodial WaaS (Wallet-as-a-Service)** like Fireblocks or Coinbase requires more backend integration and handling of webhooks, API calls, and perhaps running middleware servers. However, those services might save effort elsewhere (like not needing to implement our own crypto-to-fiat conversions or compliance integration, as they might provide those).

- *Security and Trust:* In a self-custody model, security lies largely on the user's side and the robustness of the client app. There's no central honeypot of keys to hack – which is good. And there's no need to trust us with funds; users may prefer that (no Mt. Gox risk of us losing funds). But users might also fear managing keys – if they mess up, the app can't recover their funds easily (aside from social recovery measures we add). With a custodial model, users must trust the platform (and any third-party custodian) with their money, which is a **higher burden of trust**. We'd need to earn that trust via strong security, possibly audits or showing that we use a reputable custodian. From the platform's perspective, custodial means **we are liable** if anything goes wrong (hacks, errors). Self-custody shifts liability to the user (except for providing correct software). Many fintech startups avoid custody at start due to that risk.

- *Regulatory Impact:* As elaborated, going custodial likely requires getting regulatory approval (or a regulatory sandbox) in Brazil. Non-custodial can launch without immediate licenses because it's more like providing software. The **compliance overhead** with custodial is far greater: we'd need an AML officer, routine reporting, maybe capital requirements depending on regulation. Using a third-party custodian might offload some requirements (for example, Coinbase Custody is a regulated custodian – if users directly contract with Coinbase through our app, perhaps certain obligations are on Coinbase). But likely the platform would still be seen as the service provider to the user. So from a startup agility viewpoint, **non-custodial = easier to launch quickly**. The trade-off is maybe a slightly less seamless UX (like requiring external wallet or extra steps for signing).

- *User Experience:* Custodial wallets can offer a very smooth UX – similar to how an exchange app or a neobank works. Users have an account, and they can recover access by contacting support if needed. Transactions can be abstracted (the app just shows "sent" without the user fiddling with private keys or approvals, beyond maybe a 2FA). Non-custodial wallets by default have that "friction" of seed phrases and external confirmations. But modern solutions have narrowed that gap: social login wallets and smart contract wallets can feel nearly as smooth as custodial. For example, Coinbase's WaaS and Web3Auth both aim to remove the seed phrase from the experience [73] [14]. So UX is becoming more a function of how well we implement rather than pure custody vs not. One thing to note: if the user forgets their password in a non-custodial scenario (and they didn't save a seed), recovery flows can be complex (maybe involve sending a notification to guardian emails, etc.). A custodial platform can just verify identity and restore access – which many users find comforting.

- *Scalability and Performance:* Running everything in-house (like operating our own full nodes for multiple blockchains and our own signing infrastructure) can be resource-intensive. Each Ethereum full node, for example, can require significant memory and storage, plus devops to keep it synced and up-to-date with forks/upgrades. If we support many chains, that multiplies. Using cloud providers (Infura/Alchemy or a custodian's infrastructure) scales better initially (they handle scaling nodes as our user count grows). However, relying fully on third parties can lead to bottlenecks or costs that scale with usage (Infura has rate limits or costs per call). A hybrid approach is often used: maybe run a lightweight node for certain things (for verifiability) but still use an API for heavy data. For instance, we might run a Bitcoin node to independently verify deposits, but use an API for Ethereum data where speed is more crucial. There's also the concept of **light clients** or using the user's device as part of the network (e.g. using WalletConnect's network to relay to user's wallet). These reduce our infra load at the expense of some latency perhaps.

- *Feature Control:* Running our own infrastructure gives maximum control. For example, if we run our own Ethereum node, we can capture every event and customize how we index and store them (maybe we want to attach agent IDs to certain internal logs). We can also implement custom features like our own mempool monitoring to front-run certain conditions (like if an agent schedule a payment at certain time). If we fully rely on a third-party custodian, we might be limited by their API – if they don't support a certain DeFi interaction or new token standard, we'd have to wait for them to add it. Building in-house means we can be more innovative, but it's more work and requires specialized blockchain engineers.

- *Cost:* Initially, using third-party services can be cheaper (you pay per use, and you avoid large fixed costs). But if we scale to many transactions and users, those API/custody fees might add up. Running your own nodes is costly in dev time but once up, marginal cost per transaction is low. Custodians usually charge either a monthly fee or per transaction/asset under management. For example, Fireblocks might charge a base fee plus volume-based pricing. For a startup, those costs can be significant. Non-custodial with user's own wallets means we don't pay for their transactions (the user does, via gas). If we sponsor gas or integrate heavily though, we might still incur some costs (like if using a Paymaster to sponsor transactions, we pay those fees).

Full Node vs API vs Hybrid: Another architectural decision is whether to run a **full blockchain node** or use a **hosted service/API** for blockchain interactions: - Using APIs (Infura, Alchemy, Moralis etc.) is quick and avoids maintenance. But it introduces a dependency – if that service goes down or has an outage (or gets blocked, etc.), our app could be crippled. Also, privacy: every query about user addresses goes to that third

party, which may not align with user privacy expectations. - Running full nodes gives independence and potentially security (we verify everything ourselves). But it requires maintenance – e.g. Ethereum nodes need to be updated for every hard fork, archived nodes need lots of disk (unless we use light clients). Possibly we'd want to run a node in Brazil for low latency. - A **hybrid** approach can mitigate risk: e.g., query multiple sources and cross-verify. Or use APIs for speed but have a fallback self-hosted node to failover to. Or use APIs for certain data (like historical token transfers queries) but use local components for critical tasks (like constructing and broadcasting transactions – broadcasting through multiple nodes for redundancy). - There's also the aspect of **indexing and caching**: A major part of showing a good UX (transaction lists, etc.) is having data readily available. We might need to run a database of processed blockchain data (especially if doing analytics like average cost basis for trades, etc.). We could use something like The Graph or our own ETL process to index relevant smart contract events. If we lean on providers, ensure they have endpoints for what we need (some provide account history, etc., as noted). If not, we might ingest raw data into our Firestore/Postgres to augment the chain info with our application context (like linking an on-chain tx hash to an agent proposal ID).

Offline vs Online Operations: With a custodial or integrated approach, the app can do things even when user is offline (like an agent could execute a scheduled trade at 3am on user's behalf). In non-custodial, if it requires user's signature, the user's device must come online or the user must have delegated permission to an agent key. Account abstraction could allow some pre-authorized actions (session keys). But typically, self-custody is a bit less automatic. That's a trade-off: custodial can have a more "set it and forget it" automation (like a standing order to DCA every month could be executed by the server). Non-custodial would require maybe a smart contract with built-in automation or rely on user's agent keys. However, EIP-4337's concept of **Bundlers and alternate key permissions** might help. Still, it's easier if the service just does it with custody.

Kill-switch and Emergency Control: The existing security spec mentions *"kill-switches"* – presumably the ability to halt all agent actions in an emergency. If we custody funds, a kill-switch might also freeze all crypto transfers platform-wide (maybe by turning all wallets to withdrawal-disabled state temporarily). If non-custodial, a kill-switch can prevent the app from initiating any agent transactions, but ultimately the user still has their funds and could move them externally if they wanted (which is fine – actually that's a user right in self-custody). So from a risk perspective, *in a hack scenario*, custodial is higher stakes (we'd freeze everything if an incident occurs, alert authorities, etc., as a kill-switch measure). Non-custodial, a hack of our system might not directly lead to loss of user funds (unless it's a smart contract vulnerability or an agent incorrectly approved a malicious transaction), so kill-switch might be about stopping agents from making suggestions or halting UI. Overall, architecture with smaller blast radius (non-custodial) is safer for us.

Future Extensibility: If we foresee expanding services (like offering a yield on crypto, or integrating with DeFi protocols), a custodial model would mean we have to implement those (or integrate more APIs). A non-custodial model allows users to connect to any DeFi themselves (outside the app) but if we want to facilitate it within app, we'd need to integrate DEX APIs, etc. Smart contract wallets could allow meta-transactions enabling easy DeFi use from the app though. It's a strategy question: do we want to become like a crypto bank (holding assets and perhaps lending or staking them), or remain more of an assistant that connects users to crypto ecosystem? The trade-offs align with those strategic directions.

In conclusion: - **Non-custodial/embedded**: Quick to market, fewer regulatory hurdles, lower direct liability, but must solve UX challenges and can't "hold" users' hand as much (user has more responsibility). Aligns with decentralization ethos and reduces need for heavy licensing. Possibly a better initial step to test

demand. - **Custodial/WaaS**: More seamless for mainstream users, opens possibility of revenue streams (custody fees, trading fees), and deeper integration (like offering a unified fiat+crypto experience). But requires heavy compliance lifting, security burden, and engineering on backend. Could be phase 2 once platform matures and can invest in those requirements. - **Full in-house vs reliant on external infra**: Leaning on external at start conserves resources and taps into specialized expertise (e.g. Fireblocks' security, Infura's robust nodes). Over time, as user base grows, evaluating moving parts in-house for cost savings or independence may make sense (like running our own nodes or even building a custom custody solution if we want to differentiate – though building custody from scratch is not recommended due to complexity and risk).

The architectural approach can also be mixed: for instance, we could use **Coinbase WaaS for key management** (so no seed phrase, MPC security), but still treat it as non-custodial from a product standpoint (since Coinbase doesn't control the whole key, user does via device). That might give a sweet spot: great UX, high security, and possibly no need for us to get a license since technically we never can move user funds unilaterally (Coinbase plus user's device needed). These nuances show it's not strictly binary – you can design models that borrow strengths of each side (these are often called **"semi-custodial" or "collaborative custody"** models).

Ultimately, the trade-offs will also consider **company expertise** (do we have blockchain devs to run nodes? If not, managed services are better) and **user trust**. Given Obsidian FinOps markets itself on compliance and trust, using well-known secure solutions (like Fireblocks or Coinbase) can bolster user confidence (*"my crypto is secured by Coinbase's infrastructure"*). On the other hand, tech-savvy users might prefer knowing they hold their keys (trust minimized). Perhaps offering both modes could even be a thing: e.g. default to an MPC wallet where user holds one shard (non-custodial-ish) but if a user doesn't want that responsibility, maybe offer to custody it fully (with proper agreements). However, offering both could confuse users and complicate development – likely we choose one primary path at start.

In summary, **embedded non-custodial wallet integration** is simpler and safer for a startup phase, whereas **full custodial integration** offers more control and potentially revenue (like trading fees, earning interest on float) but at the cost of becoming a quasi-financial institution. The architecture should be chosen with these trade-offs in mind, possibly starting lean and gradually moving to a more involved architecture once the user base and regulatory clarity grow.

9. Extending the Existing Security Model (Change-Set Ledger, Policies, Kill-Switches, HITL) to Crypto Flows

The Obsidian FinOps platform's security model is built around **transparency, control, and layered approvals** – every change goes through a change-set ledger with human-in-the-loop oversight [65] [43]. We need to apply these same principles to cryptocurrency transactions to maintain the platform's promise of safety and compliance. Here's how each aspect can extend to crypto:

Change-Set Ledger for Crypto Transactions: Just as the system logs every proposed state change (like a new expense entry or re-categorization) before it's committed [43] [44], any crypto transaction (send, swap, etc.) should first be recorded as a change-set event. For example, if an agent wants to execute a trade, it doesn't immediately broadcast to the blockchain. Instead, it creates a record: *"Agent X proposes sending 0.5 ETH to address Y (perhaps an exchange deposit or a payment), with the following reason…"*. This proposal is

inserted into the change pool/ledger with a unique ID and metadata (amount, asset, to/from addresses, agent or user initiating, timestamp). This ledger entry is immutable and auditable, forming the on-chain/off-chain link – even if the blockchain tx eventually happens, we have an internal ID to tie it to. Only after passing all checks does the platform actually construct and sign the blockchain transaction to execute it. This ensures **traceability**: later on, we can answer "who authorized this on-chain transfer and why?" by looking at the change-set record [74] [75]. If something goes wrong (say funds sent to wrong address), we have the forensic data to see if it was user error, agent error, or malicious.

Policy-as-Code Enforcement: The platform uses policies and rules to automatically vet change-sets [76]. We should write specific policies for crypto actions. For instance: - **Value Limits:** A policy might be "if a transaction > R$X is proposed, require CFO approval (for business accounts) or user's biometric re-confirmation (for personal)". This could be encoded so that the change manager service automatically marks such a change-set as requiring an extra approval step (maybe an admin user or a secondary confirmation from the user via email/OTP). - **Address Whitelists/Blacklists:** We could integrate with sanctions lists or the platform could let users define "trusted addresses". A policy can check: *Is the destination address on a known blacklist (e.g., OFAC sanctioned, or our internal fraud list)?* If yes, block the proposal or flag for manual review. Conversely, if it's whitelisted (e.g., user's own exchange account), maybe skip certain checks or expedite. The Fireblocks platform for example allows defining such rules [10]; we can implement our own or utilize those features if using such a service. - **Agent-specific Policies:** For each AI agent, since they each have budgets and allowed actions, we enforce those on crypto too. E.g., the "Investment Agent" might be allowed to propose trades up to $1000 per week. If it exceeds, the policy engine rejects the proposal immediately or escalates it. Another policy: *Budget Agent can't transfer assets out of the user's wallets entirely*, maybe it can only move between user's own accounts (like convert between currencies). Policies align with the *"clear spending policies and monitoring"* concept highlighted for agents [77]. - **Rate Limits and Anomaly Detection:** The platform could have policies for unusual activity: e.g., if an agent proposes 5 crypto transactions in a row that empty the wallet, flag that as anomaly (maybe the agent is malfunctioning or compromised). The policy could invoke the kill-switch automatically in such a scenario.

These policies would be coded in the change validation logic, so before a crypto change-set is approved (even by a user), it passes these gates. If any policy fails, the system could either auto-reject or require a superuser override.

Human-in-the-Loop (HITL) Approvals: The architecture mandates that by default no significant financial change happens without user approval [65]. This should absolutely apply to crypto, arguably even more so given the irreversibility of blockchain transactions. So the UX will present any agent-initiated crypto transfer to the user for confirmation. The user approving that in the app is akin to "signing off" on that transaction (the platform will then proceed to sign the actual blockchain tx with the keys). Even for user-initiated transactions, if they're done through the UI, one might consider a secondary confirmation if high value (like how some banking apps ask "Are you sure you want to transfer this large amount?"). But likely user-initiated we treat as final (with perhaps device biometric confirmation as the signature). For agent ones, definitely HITL. Over time, if the user opts for more autonomy, we could let certain known agents auto-execute within limits (the user could set rules like "Auto-invest up to $100 per week in ETF token without asking me"). The system would then treat those as pre-approved by a standing approval, but still log them and maybe send a notification. If any auto-action occurs, it should still inform the user after the fact (transparency).

Kill-Switch Mechanisms: A kill-switch is a manual or automatic trigger that stops all outbound actions. We want a kill-switch for crypto that an admin or the user can hit if something's wrong. For example, if the user

suspects their account is compromised, they hit “panic button” which: - Immediately halts any in-progress change-sets (don't execute them). - Temporarily disables agents from proposing more changes. - Freezes custodial wallets (if custodial) – e.g., flips a flag in Fireblocks to disable the wallet API [24]. - If using smart contract wallets, maybe the kill-switch triggers an on-chain pause (if we built that in), or a guardian key that can temporarily reject all ops. - Essentially puts the crypto module in read-only mode until re-enabled.

From admin side, if we detect systemic issue (like a bug causing wrong addresses or a hack), an admin kill-switch might do similar globally. The architecture audit stressed the importance of making sure nothing bypasses the checks [78] – a kill-switch is the ultimate check to ensure control.

Audit Logging and Traceability: Every step in a crypto transaction lifecycle should be logged. The change-set ledger provides an immutable record linking who (agent or user) initiated and who approved [44]. We should also log when the blockchain transaction is actually broadcast and confirmed (with the tx hash). Possibly integrate that back into the ledger entry (update status to executed with tx hash and block number). This way, if an issue arises – say a transaction was executed but the user claims they didn’t approve – we have an audit trail: user X clicked approve on date Y (maybe with an IP address/device info logged as well), then tx was sent. This non-repudiation is important for compliance and debugging. It also helps in **dispute resolution** – the audit pointed out the ledger enables dispute resolution [79]. For example, if an agent’s decision is questioned (“why did the AI send 0.1 ETH to this address?”), we look at logs: see agent rationale, user approval, etc. If it was a mistake by AI, perhaps we refund or adjust budgets accordingly. If it was user-approved but user regrets, at least we can show they did approve (educational for them).

Agent Accountability and Wallet Usage Statements: The audit envisioned showing *“statements of agent wallet usage”* to the user [80]. For crypto, this could mean each agent gets a sub-accounting of how it spent crypto. E.g., *Agent Investment used $50 of your crypto budget this month (bought 0.01 BTC)*, *Agent Bills paid 0.005 BTC for your cloud server invoice*, etc. We can produce periodic summaries where each agent’s actions are tallied, reinforcing that idea of economic actors. This not only provides transparency but helps users trust the AI – they can see tangible records of what each agent did with their money, just like a mini bank statement for each agent’s “wallet”. This feature aligns perfectly with the accountability vision [81] and should extend to on-chain flows.

Multi-Sig and Multi-Party Approvals: The platform’s security can also extend physically into how transactions are authorized. For example, if we have an internal key controlling part of an approval (like the platform guardian key in a smart wallet scenario), that key’s usage could itself be guarded by requiring multiple internal sign-offs. Fireblocks provides an option where admin and co-admin both must approve a large transfer (owner and admin roles) [52] [23]. We can simulate that in our platform by requiring, say, a senior admin to countersign any transaction above a threshold even after user approval. This would be invisible to user but an internal policy (belt and suspenders approach). It’s more relevant if custodial; if non-custodial with user in control, then user approval is the final word.

Integration with Existing Security Services: The architecture possibly already has security services like anomaly detectors or simulating changes impact. We can integrate chain analytics here: e.g., simulate the impact of a transaction on budgets or forecasts (if I send this much Bitcoin, will it upset my budget?). Or integrate with a service to simulate the on-chain transaction before sending (like the Noves Foresight API that simulates and describes a transaction to detect risks [23]). That could be an automated step: before an agent-proposed complex transaction (like interacting with a DeFi contract), simulate and show the user “this

will result in you receiving X tokens, with Y gas fee" to avoid any hidden surprises (which sometimes happen in DeFi). It's analogous to how the FinOps might simulate a budget change's effect on cash flow before committing.

Continuous Monitoring and Alerts: After a crypto transaction is executed, the platform can continue to monitor for outcomes, especially if something requires follow-up (like awaiting confirmations or if an on-chain event triggers something). For example, if a transaction remains unconfirmed for too long (maybe stuck due to low gas), an agent could alert or take action (resubmit with higher fee). Or if a large outgoing transfer happens (even user-initiated), the system could send a push notification/email separate from the app (like banks do: "You sent X – if this wasn't you, contact support immediately"). These alerts augment security by enabling quick reaction if something unauthorized did slip through.

Fail-Safe Design: In any complex flow (with agents and automation), we consider fail-safe defaults – ideally "off" for money movement. That is, if any component of the system is uncertain or down, it should default to not sending money rather than sending without checks. For instance, if the policy engine fails to respond, the change manager should treat that as a failure and not approve the transaction. If the user's approval status can't be verified (say database glitch), we don't assume it's approved. These little design choices prevent edge-case mishaps.

Testing with Small Amounts and Kill-Switch Drill: We might incorporate in the UX something like a "test send" feature where an agent or user can do a trivial small transaction initially to a new address and ensure everything works, before larger. Not a requirement, but could be nice. Also, we could simulate kill-switch drills – e.g., provide guidance to the user on how to engage emergency freeze if needed (maybe not expose it too prominently to not scare them, but in support docs).

By weaving the crypto functionality into the existing rigorous security framework, we ensure that introducing this new capability does not compromise the platform's core promise of controlled autonomy. Instead, crypto flows become first-class citizens in the system: every on-chain action is as **observable, governable, and reversible (to the extent possible)** as a normal off-chain action. Of course, on-chain transactions can't literally be reversed like a database change, but our ledger and possibly smart wallet design could allow *compensation transactions* (like sending back funds) if, say, an agent error is detected – and those would also go through approval.

In essence, we maintain the **"single audited path"** principle: every crypto transaction must go through the same funnel of proposal → validation (policy) → approval → execution with logging [48]. Nothing should ever "sneak out" directly to blockchain without that funnel. By doing so, we extend our 100% compliance approach to the crypto domain, giving users confidence that even in the wild west of crypto, our system is watching out for them at every step.

Conclusion: Through the above analysis, we've outlined how an Obsidian FinOps-like platform can incorporate a cryptocurrency wallet in a comprehensive yet secure and compliant manner. We covered the necessary technical infrastructure (from node connectivity to key storage), contrasted custodial vs. non-custodial approaches and the simplest ways to implement each (leveraging existing wallets or services), integration of crypto data into the user's financial picture, robust private key management options, the use of smart contract wallets (EIP-4337) to encode our policy logic on-chain, UX flows needed to make crypto user-friendly, Brazilian-specific compliance (tax declarations, LGPD), trade-offs of different architecture

decisions, and finally how to map our proven change-set approval and audit model onto crypto transactions. By following these guidelines and citing real-world examples and standards, the platform can evolve into a crypto-enabled financial OS without losing the trust and control that are its hallmarks, effectively becoming a **crypto wallet that is as intelligent, compliant, and user-centric as the rest of the FinOps ecosystem**.

Sources:

- Obsidian FinOps Architecture Audit – handling of agents, ledger, compliance [60] [12]
- Legal requirements for custodial vs. non-custodial wallets [2] [3]
- Coinbase WaaS and Web3Auth for easy wallet integration [14] [16]
- Covalent API for multi-chain balance/tx fetching [7] [39]
- Brazilian tax rules (IRPF thresholds, RFB reporting) [45] [67]
- Safe (Gnosis Safe) as example smart wallet with multisig [32] [33]
- Fireblocks/Scorechain for compliance and policy controls [10] [52]
- Thirdweb smart wallets and ERC-4337 features [17] [18]
- Unlock Labs case: Coinbase WaaS + Google login UX [26] [34]
- Platform audit on agent wallet usage accountability [80] and change-set approvals [43].

---

[1] [2] [3] [71] A Legal Guide to Custodial & Non-Custodial Wallets
https://www.legalnodes.com/article/custodial-non-custodial-wallets

[4] [11] [69] [70] Travel Rule Crypto in Brazil by the BCB [2025] - Notabene
https://notabene.id/world/brazil

[5] [6] [49] Custodial vs. Non-Custodial Wallet: Differences and How to Choose
https://cheesecakelabs.com/blog/custodial-vs-non-custodial-wallets/

[7] [8] [23] [38] [39] [40] [52] Crypto Wallet APIs - Track balances, transfers, and risk
https://marketplace.quicknode.com/explore/crypto-wallet-apis

[9] [10] Define Travel Rule Policies - Introduction - Fireblocks
https://developers.fireblocks.com/docs/define-travel-rule-policies

[12] [13] [35] [36] [37] [41] [42] [43] [44] [48] [60] [61] [65] [66] [72] [74] [75] [76] [77] [78] [79] [80] [81] Final Architecture Audit_ Obsidian FinOps 100% Compliant OS.pdf
file://file-A5LDdWvgMrBvCtLFduCEeJ

[14] [15] [25] [27] [73] Coinbase launches wallet-as-a-service for businesses
https://cointelegraph.com/news/coinbase-launches-wallet-as-a-service-for-businesses

[16] Documentation | Web3Auth
https://web3auth.io/docs/

[17] [18] [19] [20] [21] [28] [29] [30] [31] Introducing Wallet SDK: The complete web3 wallet toolkit
https://blog.thirdweb.com/introducing-wallet-sdk-the-complete-web3-wallet-toolkit/

[22] Fireblocks' New Developer APIs: Build on Blockchain Without ...
https://www.fireblocks.com/blog/fireblocks-new-developer-apis-build-on-blockchain-without-complexity/

24 API Communication
https://ncw-developers.fireblocks.com/docs/api-communication

26 34 64 Unlock Labs Integrates Coinbase Wallet as a Service for Embedded Wallets
https://unlock-protocol.com/blog/unlock-labs-integrates-coinbase-wallet-as-a-service-for-embedded-wallets

32 33 62 Video: Web3 Wallets - Gnosis Safe Wallets - Web3 Wallet Security Basics
https://updraft.cyfrin.io/courses/web3-wallet-security-basics/signer-basics/gnosis-safe-wallets

45 Brazil Crypto Tax: Ultimate Guide 2025 - CoinLedger
https://coinledger.io/blog/brazil-crypto-tax

46 Guide to Crypto Taxes in Brazil for 2025 - TokenTax
https://tokentax.co/blog/crypto-taxes-brazil

47 Brazil Introduces New Capital Gains Tax On Cryptocurrencies - KoinX
https://www.koinx.com/tax-guides/capital-gains-tax-on-cryptocurrencies-brazil-update

50 51 53 54 55 56 57 58 59 ERC-4337 & Account Abstraction: A Comprehensive Overview - Hacken
https://hacken.io/discover/erc-4337-account-abstraction/

63 mirror.xyz
https://mirror.xyz/_next/image?url=https%3A%2F%2Fimages.mirror-media.xyz%2Fpublication-images%2FotIfC7ESSqjLLPSoktrkc.png&w=3840&q=75

67 Crypto assets: aspects about the legal framework, taxation and ...
https://www.dpc.com.br/crypto-assets-aspects-about-the-legal-framework-taxation-and-reporting-requirements/?lang=en

68 Brazil Tax Authority Requires Reporting All Bitcoin Transactions ...
https://cointelegraph.com/news/irs-brazil-requires-reporting-all-bitcoin-transactions-starting-today