Sail Protocol

Onchain Separately Managed Accounts Run by Agents

Alejandro Dopico
Cofounder and CEO

Contributors: Alvaro Alonso, Andrés Sarría

June 2026

`https://sail.money`

Abstract

Sail Protocol[1] is a protocol for onchain separately managed accounts (SMAs), implemented for the Ethereum Virtual Machine. Capital is held in a self-custodial Safe the owner controls; a designated manager—typically an autonomous agent—executes transactions within a mandate enforced by smart contracts on every dispatch. The mandate is a set of user-deployed Solidity permission contracts registered against the account. The manager's signature names one registered permission as the authorizer for each dispatch; the kernel evaluates that permission via `staticcall` under a gas cap and forwards the call to the Safe only if it returns true. Because permissions are arbitrary Solidity, any DeFi primitive can be expressed as a permission: the protocol covers, by construction, everything DeFi can do. Protocol fees are bounded by immutable constitutional caps. Anyone can deploy SMAs, write permissions, and operate as a manager.

[1] `https://github.com/sail-money/protocol`

Disclaimer

This paper is published for general information purposes only and describes the Sail Protocol smart contracts at a technical level. It does not constitute, and must not be relied on as, investment, financial, tax, accounting or legal advice, nor as any recommendation or solicitation to buy, sell, deposit, delegate, or otherwise transact in any digital asset, token, or smart contract position. Nothing in this paper should be construed as an offer of securities or of any other financial instrument.

The terminology used throughout this paper is used solely in a descriptive and functional sense to refer to the mechanical behaviour of the Sail Protocol smart contracts. In particular, the terms "separately managed account", "SMA", "account", "mandate", "manager", "owner", "custody", "self-custodial", "permission", "fee", "management fee", "performance fee", "portfolio", "strategy", "marketplace" and any cognate terms are used in a descriptive and functional sense to refer to the mechanical behaviour of the Sail Protocol smart contracts. None of these terms is intended to denote, and none should be construed as denoting, a regulated financial instrument, a security, an investment-advisory or asset-management relationship within the meaning of any applicable securities, banking, or financial-services regulation, a deposit, a custodial arrangement, a collective investment scheme, or any other regulated product, activity or status. Any resemblance between the vocabulary used herein and concepts defined under applicable laws or regulations is incidental and does not reflect the legal characterisation of the protocol, its components, or the activities of its users.

The Sail Protocol source code is provided "as is", without warranty of any kind, express or implied, including without limitation any warranty of merchantability, fitness for a particular purpose, non-infringement, or accuracy. "Sail" is a trademark of its owner; no licence to use it is granted by this paper, and any third-party use of the mark remains subject to applicable trademark law.

Any returns, rates, or other figures referenced in this paper—or in related communications—are not guaranteed and are for illustrative purposes only. Use of the protocol may be restricted in certain jurisdictions, and users are solely responsible for compliance with all laws and regulations applicable to them. The views and statements expressed in this paper are current as of its date of publication and are subject to change without notice and without any obligation to update. To the maximum extent permitted by applicable law, the authors disclaim any and all liability for any direct, indirect, incidental, consequential or special losses, damages, costs or claims arising from or in connection with any reliance on this paper or any use of the Sail Protocol smart contracts.

1 Introduction

Personalized investment management has historically been an institutional product. A separately managed account (SMA) is the structure that supports it: capital is held in an account titled to the owner, while a designated manager executes investments within bounds set by a mandate. Unlike pooled vehicles, the assets remain separately custodied for each client; the manager has bounded authority, not full discretion; the mandate can be revised, narrowed, or revoked at any time; and performance is attributed per account. These properties made the SMA the standard structure for institutional and large-scale allocation. They have also kept it inaccessible: account minimums of $100,000 to $1,000,000 are typical, because the cost of custodians, legal infrastructure, and reconciliation is prohibitive at smaller sizes.

Decentralized finance has not replicated the structure. The dominant managed-account pattern in DeFi is the pooled vault: depositors mint shares of a strategy operated by a single manager, giving up account-level custody, sharing attribution with every other depositor, and inheriting the strategy's full risk profile. Self-custodial smart wallets preserve custody but offer no native concept of a manager bounded by a mandate. The category between them—an onchain SMA primitive—has been missing.

Autonomous agents make the gap urgent. AI systems can now execute trades, rebalance portfolios, and respond to market conditions continuously. To act onchain on behalf of capital they do not own, they need two properties at once: the ability to sign transactions, and enforceable bounds on what they may do. Existing approaches force a choice—give the agent a private key and surrender custody, or run it through an off-chain co-signer and lose machine speed. The legal mandate that governs traditional SMAs cannot resolve this: an agent executing continuously cannot be governed by paperwork it cannot read. The bounds must be code, evaluated on every transaction, revocable in a single block.

Code-enforced mandates also change what a strategy can be. Traditional asset management is static: one template, applied uniformly across a book of accounts and rebalanced on a human schedule. An agent operating inside an onchain SMA enables *dynamic asset management*—management that is continuously sensitive to each account's own variables, calibrating to its balance, risk tolerance, time horizon, and liquidity needs in real time rather than fitting it to a shared template. This is a new category: an agent managing a thousand accounts runs a thousand strategies, sharing one set of infrastructure, one set of permission templates, and one fee model, and the marginal cost of one more account is one more agent execution.

Sail Protocol expresses the SMA relationship as a minimal onchain primitive. A minimal trusted kernel handles account instantiation, permission registration, manager dispatch, and fee splitting. Everything else—what permissions express, how fees are calculated, how NAV is reported, which venues are supported—lives in user-deployed contracts outside the trusted core. The kernel is auditable in isolation. Each permission is auditable in isolation. The protocol supports any DeFi primitive, current or future, without kernel changes.

2 Protocol Model

2.1 Three Roles

Sail separates three roles that exist implicitly in any managed-account structure.

The role of transaction submitter—the address that pays gas and submits the manager's signed

Role	Authority	Held by
Owner	Holds the Safe. Custodies the SMA's capital. Always self-custodial.	The account owner, whose capital it holds.
Permission Signer	Authorizes the mandate. Signs registration, configuration, and revocation of permissions via EIP-712.	The Owner, or a separate signing key or multisig.
Manager	Executes transactions within bounds. Cannot exceed what the registered permissions allow.	An autonomous agent or a human operator; the signing key may be an EOA, multisig, or MPC wallet.

Table 1: Core protocol roles.

dispatch—is intentionally not an authority role. Any address may submit; authority derives from the Manager's signature, the registered permissions, and the kernel's evaluation. This makes the protocol natively compatible with relayers, paymasters, and ERC-4337 bundlers.

2.2 The Mandate

In Sail, the mandate is the set of permissions registered for an SMA—contracts implementing `IPermission` that define what the Manager is authorized to do. The Safe is the account the mandate applies to; it is not itself part of the mandate.

When the Manager submits a transaction, the signature names one registered permission as the authorizer. The kernel calls `evaluate()` on that permission alone and dispatches the call to the Safe only if it returns true. If the manager attempts to swap on an unallowed router, transfer to an unallowed recipient, borrow above the configured LTV, or call any function outside the registered permission set, the transaction reverts before any state change occurs. Figure 1 shows the full relationship.

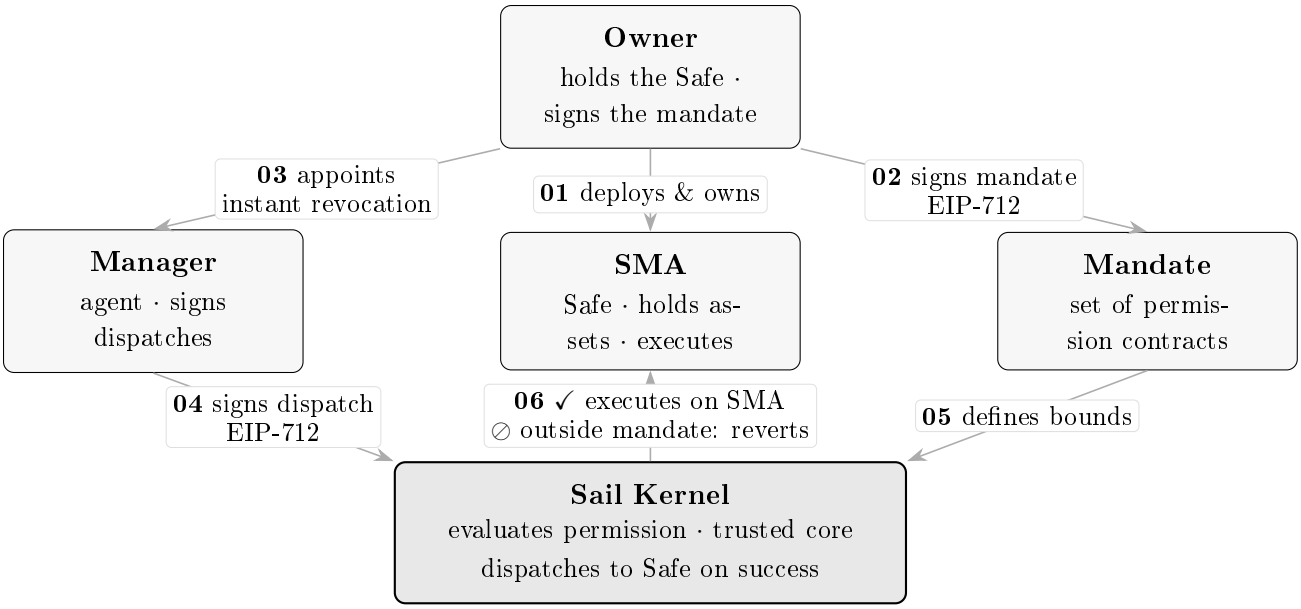

Figure 1: The Sail Protocol model. The Owner deploys and owns the SMA (a Safe), signs the mandate (the set of registered permission contracts), and appoints the Manager. The Manager signs a dispatch message (EIP-712) naming one permission as the authorizer; the Mandate defines the bounds the Kernel enforces. The Sail Kernel evaluates the named permission via `staticcall` on every dispatch and executes the call on the SMA only if it returns true—otherwise the dispatch reverts. Transactions execute on the SMA; the Kernel never holds assets.

3 Architecture

3.1 Components

The protocol comprises a minimal trusted core, plus a default fee policy and a starter set of permission templates. The trusted core is what users must trust to use Sail at all. The default policy and templates are what users must trust if they choose them; alternatives can be deployed by anyone without protocol-level changes.

The kernel never holds custody of assets. The Safe holds the capital. The kernel holds only the permission list, principal accounting, and fee policy reference for each account.

3.2 Batch Dispatch

The protocol provides a second dispatch path for multi-step strategies that require atomic execution. `dispatchBatch()` accepts an ordered array of calls and a single batch permission—a contract implementing `IBatchPermission`—that validates the entire call sequence before any execution begins. Each call executes via the Safe module path in order; any subcall failure reverts the entire batch atomically.

This path is designed for strategies requiring temporary ERC-20 approvals: the batch permission validates an approve–execute–reset sequence as a unit, enforcing that the approval amount matches the consumed amount and that the cleanup reset is present. It is equally available for any multi-step strategy where atomicity is required.

Component	Role
Trusted core	
SailKernel	Account registration, permission registry, EIP-712 signature verification, manager dispatch via Safe modules, fee collection, principal tracking.
SailGovernance	Protocol parameter governance behind a 48-hour timelock; emergency pause with auto-expiry and cooldown; two-step governance transfer; trusted Safe factory, singleton, proxy-codehash, and fee-policy allowlists.
TimelockController	Standalone timelock, deployed separately and injected into SailGovernance, which validates it at construction (Section 7).
SafeModuleEnabler	Stateless helper that enables the kernel as a Safe module during account creation.
MandateFactory	Untrusted UX orchestrator. Bundles permission configuration, registration, replacement, and detachment into single transactions. Holds no protocol-level privileges.
ConfigurablePermission	Abstract base for shared multi-tenant permission templates. EIP-712 domain, per-account nonces, ECDSA and ERC-1271 signature verification.
Interfaces	`IPermission`, `IBatchPermission`, `IConfigurablePermission`, `IFeePolicy`, `IOracle`, `IPermissionIntrospection`, `IAgentIdentityResolver`.
Swappable defaults	
StandardFeePolicy	Reference fee policy: management fee on AUM, performance fee above a per-account high-water mark. Any account may register a different `IFeePolicy` instead.
Permission templates	Reference multi-tenant permission templates covering common DeFi primitives; an account may instead register any contract that implements `IPermission` (Section 4.4).

Table 2: Protocol components.

The manager's signature commits to the permission address, the exact call array, a nonce, and a deadline. Batches use a nonce namespace separate from single dispatch and are bounded in length. Batch dispatch follows the same selective authorization model as single dispatch: one named permission evaluates the entire batch.

4 Permission System

4.1 The IPermission Interface

A permission is a contract implementing a single interface:

```
interface IPermission {
    function evaluate(bytes calldata txData, Context calldata ctx)
        external view returns (bool);
    function discriminator() external view returns (bytes32);
}

struct Context {
```

```
    address account;         // the Safe
    address manager;         // the delegated signer
    address submitter;       // msg.sender of the dispatch (may be a relayer)
    address target;          // call target
    bytes4  selector;        // call selector
    uint256 value;           // msg.value
    uint256 blockTimestamp;
    uint256 blockNumber;
}
```

`evaluate` returns true if the transaction is allowed by this permission, false otherwise. `discriminator` returns a hash describing the (target, selector) tuple the permission gates, enabling tooling to group and short-circuit permissions for high-volume use cases.

4.2 Evaluation Semantics

The kernel's evaluation of permissions has four properties that together constitute the protocol's runtime guarantee.

Static evaluation. Permissions are called via `staticcall`, which prohibits state mutation. A permission cannot modify any contract's storage during evaluation. Reentrancy through the permission surface is structurally impossible.

Gas isolation. Each permission is called with a fixed gas cap of 150,000. A permission that exceeds its cap reverts and is treated as returning false. A pathological permission cannot deny service to the kernel or consume the manager's gas budget beyond the cap.

Selective authorization. The manager's signature names one registered permission as the authorizer for the dispatch. The kernel evaluates that permission alone—no other registered permissions are consulted. The mandate is the union of registered permissions; each dispatch selects one as its authorizer. This enables unrelated templates to coexist on one account: a swap permission, a borrow permission, and a transfer permission can all be registered, and each call selects the appropriate authorizer without the others falsely denying it.

Fail-closed. Any permission that reverts, runs out of gas, returns malformed data, or returns false causes the entire dispatch to revert. The default behavior of a buggy permission is to deny, not to allow. An account with no registered permissions cannot dispatch at all.

4.3 Full Expressiveness

Because permissions are arbitrary Solidity contracts, the protocol does not bound what a permission can express. The alternative path is a fixed-grammar mandate language: a finite set of conditions and parameter kinds, interpreted by a constraint virtual machine inside the kernel. That approach trades expressiveness for predictability—and pays for it on every new DeFi pattern, which requires extending the grammar, updating the VM, and shipping a kernel upgrade.

Sail inverts the trade-off. The kernel knows nothing about DeFi venues. It calls `evaluate()` on a permission contract and respects the answer. The structural guarantees—`staticcall`, gas cap, fail-closed, selective authorization—protect the kernel from the permission; everything the permission expresses inside that envelope is the author's responsibility. Adding a new DeFi integration to Sail is a contract deployment, not a protocol upgrade.

The single class of venues outside this envelope is venues whose state is partly off-chain—order books

that match off-chain, settlement engines that sign off-chain. Permissions can constrain the on-chain boundary of such venues (deposits, withdrawals, allowed sub-accounts) but cannot constrain off-chain order signing. That is a property of those venues, not of the protocol.

4.4 Example Templates

To demonstrate the permission pattern across common DeFi primitives, Sail ships a starter set of shared templates. These are not the protocol; they are demonstrations of it. Anyone may deploy additional permission contracts for any DeFi venue; the kernel will register and dispatch through any contract that implements `IPermission`.

Template	Gates
SwapPermission	DEX swaps. Router and token allowlists, size cap, output paid to the account, slippage floor against an independent price oracle.
SwapPermissionNoOracle	DEX swaps without an external oracle. Router and token allowlists, size cap, slippage floor against the reference pool's own live price (V2 reserves / V3 `sqrtPriceX96`).
BorrowPermission	Lending borrows. Protocol and asset allowlists, size cap, position credited to the account, optional LTV ceiling against oracles.
DepositPermission	Deposits into allowlisted vaults and lending pools. Token and target allowlists, size cap, position credited to the account.
WithdrawPermission	ERC-20 movements pinned to one configured recipient. Token allowlist, size cap, `transferFrom` source must be the account.
TransferPermission	ERC-20 sends to an allowlisted recipient set. Token allowlist, size cap, `transferFrom` source must be the account, native ETH rejected.
ApproveAndCallBatchPermission	Atomic approve / protocol-call / reset batch. Token and spender allowlists, amount cap, paired (target, selector), mandatory reset to zero.

Table 3: Example permission templates.

4.5 Shared Multi-Tenant Templates

The naive deployment pattern is one permission contract per account, with parameters in the constructor. This produces a clean per-instance audit surface but is expensive in gas and bytecode. Sail's recommended pattern is the shared multi-tenant template: one contract deployed once per chain, serving every account that registers it, with per-account configuration stored in mappings keyed by account address. Configuration is performed through `IConfigurablePermission`:

```
interface IConfigurablePermission is IPermission {
    function configure(address account, bytes calldata params,
        uint256 deadline, bytes calldata sig) external;
    function configureDirect(address account, bytes calldata params) external;
```

```
    function configNonces(address account) external view returns (uint256);
    function isConfigured(address account) external view returns (bool);
}
```

The `params` field is opaque template-specific calldata, decoded inside the template's configuration hook. The kernel and factory know nothing about `params`; new template shapes require zero changes to either.

4.6 Permission Lifecycle

A permission's lifecycle is governed by operations authorized by the Permission Signer through EIP-712 signatures.

Registration. The Permission Signer signs a registration message containing the permission contract address. The kernel adds it to the account's permission list and charges the registration fee (Section 6).

Configuration and reconfiguration. For shared templates, the Permission Signer signs a configure instruction containing the template-specific `params` blob. Any caller may submit the signed call; the canonical orchestrator is the MandateFactory, but it holds no special privilege. A subsequent configure with a fresh nonce clears previous state and applies the new `params` atomically.

Replacement. Sets of permissions can be replaced atomically in a single signed operation, eliminating the unguarded window that would exist between a revocation and a re-registration.

Revocation. The Permission Signer signs a revocation. Two levels exist: revoking a single permission narrows the manager's authority; revoking the manager session entirely cuts off the manager. Revocation takes effect in a single block and additionally invalidates all outstanding manager-signed dispatches that predate it.

Manager rotation. The Safe itself can rotate the delegated manager. Rotation clears all registered permissions atomically—no mandate silently transfers authority approved for the old manager to a new one—and invalidates all outstanding manager-signed messages.

4.7 Optional Extension Interfaces

Templates may optionally implement `IPermissionIntrospection`, exposing a stable identifier, version hash, metadata URI, and capability identifiers used by indexers and discovery tooling—and `IAgentIdentityResolver`, associating the manager with an external agent identity registry. Both are conventions for the tooling layer; neither is required or verified by the kernel, consistent with keeping the trusted core minimal.

5 Deterministic Deployment and Chain-Portable Accounts

The trusted core is deployed through a CREATE2 factory with chain-independent salts and identical constructor arguments on every supported chain. Identical init code and identical salts yield identical addresses: every core contract—kernel, governance, timelock, factory, fee policy, module enabler—lives at the same address on every supported chain.

Account addresses inherit the property. When an account is created, the kernel derives the CREATE2 salt by binding the caller's salt nonce together with the principals of the account:

```
boundSalt = keccak256(saltNonce, caller, permissionSigner, manager, feePolicy)
```

The Safe initializer references only the kernel and the module enabler—both chain-identical—so the same owner, permission signer, manager, fee policy, and salt nonce produce the same SMA address on every supported chain. Binding the principals into the salt also means a counterfactual address cannot be front-run with different principals: a deployment supplying a different manager or signer lands at a different address.

The consequence is chain portability: across chains that share identical core construction, both the trusted core and each SMA resolve to one address, and assets sent there reach the same account whether or not the Safe has been deployed on that chain yet. The property holds only where the construction arguments match. One such argument is the per-permission registration-fee ceiling, denominated in the chain's native token (Section 7); a chain whose native token differs materially may be deployed with a different ceiling, in which case its core and accounts resolve to different addresses. The protocol is deployed on Ethereum, Base, Arbitrum, and Unichain.

6 Fee Model

The protocol enforces two independent fee mechanisms. Each is capped by an immutable constitutional limit and tunable within that limit by governance.

6.1 Fee 1: Permission Registration Fee

When a permission is registered with the kernel, the registering account pays a flat ETH amount to the protocol treasury, identical regardless of contract size or template type:

$$\text{total fee} = \texttt{permissionRegistrationFee} \times n_{\text{permissions}}$$

The active rate (`permissionRegistrationFee`) is held in SailGovernance, set at deployment and tunable through the timelock—it is not a hardcoded constant. It is bounded by an immutable per-deployment maximum (`MAX_PERMISSION_FEE_WEI`), which the SailGovernance constructor in turn caps at a constitutional ceiling of 0.01 ETH. Excess `msg.value` is refunded. The fee is denominated in native ETH with no oracle dependency; governance retunes the active rate as ETH price moves.

6.2 Fee 2: Protocol Cut on Manager-Collected Fees

When the Manager calls `collectFees`, the kernel asks the registered `IFeePolicy` for the legitimate fee amount, then splits it:

$$\text{protocol cut} = \text{managerGrossFee} \times \texttt{currentProtocolCutBps}/10{,}000$$
$$\text{manager take} = \text{managerGrossFee} - \text{protocol cut} - \text{distributor cut}$$

The active cut is governance-tunable, bounded above by the immutable cap of 25%: the protocol can never take more than a quarter of what a manager collects, regardless of any future governance procedure. *The protocol cut defaults to zero.* Setting any non-zero cut is a separate governance decision, bounded by that cap.

The kernel does not compute the gross fee. That responsibility lies in the registered `IFeePolicy` contract, which contains the schedule.

6.3 Fee Policy Extensibility

`IFeePolicy` is an interface, not a fixed contract. The reference implementation, StandardFeePolicy, provides a management fee on AUM and a performance fee above a per-account high-water mark, using a manager-attested NAV model appropriate for self-managed SMAs where the manager and the owner are the same party (Section 8.2). Alternative policies suited to third-party allocation—trustless NAV from on-chain position adapters, fees crystallized only on withdrawal, independent attester co-signatures—can be deployed by anyone.

7 Governance

7.1 Constitutional Caps

Two bounds are fixed in the source code. No governance procedure can raise them.

- The maximum protocol cut on manager-collected fees: `MAX_PROTOCOL_CUT_BPS` = 2,500 (25%).
- The maximum permission registration fee: a constitutional ceiling of 0.01 of the chain's native token (0.01 ETH on Ethereum and ETH-denominated chains) that the immutable `MAX_PERMISSION_FEE_WEI`, set per deployment, can never exceed.

These bounds are properties of the deployed bytecode. A future governance under any control structure—multisig, token, DAO—cannot raise them. Lowering the active parameter inside these bounds remains possible. Because this ceiling is denominated in the native token and fixed in the bytecode, a chain with a materially different native token may carry a different cap—the construction difference behind the cross-chain address variation in Section 5.

7.2 Governance Parameters

The following are mutable within the constitutional caps: the active protocol cut (zero by default); the active registration fee; the trusted Safe factory, singleton, proxy-codehash, and fee-policy allowlists; and the emergency admin. Parameter mutations require a 48-hour timelock delay followed by execution. Governance transfer is itself two-step (propose and accept). Emergency pause is a separate, lower-latency channel held by an emergency admin; pauses expire automatically and a cooldown applies between invocations to prevent denial of service by spam-pausing.

The timelock is a standalone contract injected at construction, and SailGovernance validates it before accepting it: the delay must equal 48 hours exactly, governance must hold both the proposer and executor roles, and the timelock must administer its own roles, with no external account holding administrative power over them. A deployment that fails any of these checks reverts.

Governance is initially held by the team multisig. The contract is structured to support transfer to a token, DAO, or other mechanism without changes to the kernel.

8 Security Properties

8.1 Formal Guarantees

The protocol provides six guarantees, each a property of the deployed bytecode rather than of governance policy.

1. **Custody isolation.** The kernel cannot transfer Safe assets except through a manager dispatch that satisfies the named permission's evaluation. The kernel has no direct write access to the Safe outside the module dispatch path.
2. **Selective authorization.** A dispatch succeeds only if the permission named in the manager's signature is registered for the account and returns true on evaluation.
3. **Reentrancy safety.** Permission evaluation occurs via `staticcall`, which prohibits state mutation. No re-entry path exists through the permission surface.
4. **Gas isolation.** Each permission is called under a fixed gas cap; exceeding it is treated as returning false. The kernel cannot be denied service by a malicious permission.
5. **Constitutional fee caps.** Protocol cut and registration fee cannot exceed their immutable bounds under any governance procedure.
6. **Signer separation.** The Permission Signer cannot move Safe assets. The Manager cannot register or revoke permissions. The Safe Owner can always revoke the Manager.

8.2 Known Limitations

The protocol does not guarantee the following properties, which follow from its design as open infrastructure.

Permission correctness. A buggy permission is still a permission. The kernel verifies that a permission is a contract and that it returns true or false; it does not verify that the permission's logic correctly enforces what its author claims. Users are responsible for the permissions they register.

Upgradeable permissions. The kernel binds a registered permission by its address and does not re-check its code on each dispatch. An upgradeable or otherwise mutable permission can therefore change its `evaluate` behaviour after the Permission Signer has approved it, silently widening the mandate. Owners should register only non-upgradeable, reviewed permission contracts.

Manager-attested NAV. The reference fee policy uses a manager-attested NAV model: the manager reports portfolio value at fee collection time. A manager operating an SMA holding third-party deposits could in principle inflate the reported NAV when collecting fees, unlocking a fee that the kernel bounds only by the account's own balance—in the limit, approaching a full withdrawal of the account. This model is appropriate where the manager and the owner are the same party—an account operated with its owner's own capital. A third-party allocation on the open protocol warrants independent due diligence on the manager, the fee policy, and the registered permissions, or a fee policy that derives NAV without manager attestation.

Off-chain venue components. For venues whose state is partly off-chain, permissions can constrain the on-chain boundary (deposits, withdrawals, allowed sub-accounts) but cannot constrain off-chain order signing. Permissions for such venues must disclose which surfaces are enforced on-chain and which inherit venue-specific trust assumptions.

Further limitations. In-place rotation of the Permission Signer is not provided; recovery from a compromised separate signing key is by migrating the Safe to a fresh account. A registered permission is bound by its address, and its code hash is not pinned at registration—the reason the upgradeable-permission boundary above applies. Native-ETH fee collection requires a payable recipient, so ERC-20 fee assets are preferred where payability is not guaranteed. The emergency pause that halts dispatch and fee collection expires automatically and is rate-limited by a cooldown

between invocations.

9 Use Cases

Sail is infrastructure for personalized, dynamic asset management, and it serves two kinds of operator. The first is an individual delegating to an autonomous money agent: a crypto-native owner who grants an agent bounded authority over their own capital and lets it act continuously within the mandate. The second is an asset manager running dynamic strategies at scale: a distinct, continuously-calibrated strategy per account, each account bounded by its own registered permissions, all sharing one set of infrastructure. Both reduce to the same primitive—a Safe, a mandate, and a manager the kernel holds to it.

The protocol is venue-agnostic. Any on-chain primitive becomes a permission template: AMM swaps, lending deposits and borrows, LP positions, perpetual trades, restaking deposits, prediction market bets, RWA flows.

For venues with fully on-chain state, permissions enforce the full set of mandate constraints—allowlisted routers, bounded amounts, LTV ceilings, slippage tolerances, recipient checks—at calldata level, before the call leaves the kernel. For hybrid venues with partly off-chain state, permissions enforce what is enforceable onchain, and the boundary is documented per template.

The agent-operated SMA makes the model concrete. A developer building an agent strategy deploys a Safe, registers the permission templates corresponding to the agent's expected actions, configures bounds, and grants the agent's signing key as the Manager. The agent runs continuously within the registered bounds. It cannot drain the Safe, cannot redirect funds, and cannot exceed its mandate. If the agent malfunctions or is compromised, the owner revokes its permissions onchain and the agent's authority ends immediately. The custody and bounds infrastructure is the protocol's responsibility, not the application's.

10 Non-Goals

Sail's design is shaped as much by what it excludes as by what it includes. The protocol explicitly does not provide:

- **Policy authoring workflow.** Drafts, versions, curation. Off-chain authoring—Git, IDEs, SDKs, frontends—handles this. The protocol stores deployed permission addresses, not authoring metadata.
- **A constraint grammar.** Solidity inside permission contracts replaces interpreted constraint structs. There is no enum of conditions, no VM of operations. There is only the `IPermission` interface.
- **Workflow execution as a kernel concept.** A multi-step workflow is just a kind of permission: the transaction must conform to this shape. The protocol does not need to know.
- **ERC-4337 and EIP-7702 adapters in the kernel.** Peripheral adapter contracts wrap the kernel for users who want those entry points. The kernel remains transport-agnostic.
- **NAV computation.** Lives in user-deployed valuation modules and oracle adapters. Oracle choice is an ecosystem concern, not a protocol concern.

- **A registry of curators or template authors.** An off-chain concern, outside the protocol.
- **Identity primitives.** Permission modules may consult external identity registries; the kernel stays agnostic.

Each exclusion reduces what the protocol owns. The kernel owns less, by design, so that what it does own is provable, auditable, and stable for years. The off-chain side of these exclusions is served by Sailor, an open-source SDK, CLI, and local dashboard for building and operating mandated agents against the protocol. Sailor is separate from the protocol and is not part of the trusted core.

11 Conclusion

Sail Protocol expresses the separately managed account as a minimal onchain primitive. A minimal trusted kernel mediates between an Owner who holds custody through a Safe, a Permission Signer who authorizes the mandate, and a Manager who executes within bounds. Permissions are user-deployed Solidity contracts evaluated under structural guarantees—`staticcall`, gas cap, selective authorization, fail-closed. Fees split through a protocol-enforced mechanism with immutable constitutional caps. The core lives at one address on every supported chain, and so does each account. Everything beyond the kernel lives outside it.

The protocol exists because autonomous agents have made the legal mandate insufficient. An agent executing at machine speed cannot be governed by paperwork. The mandate must be code, evaluated on every transaction, revocable in a single block.

It also exists because the SMA structure has been inaccessible to most investors for a century. Onchain, with a minimal kernel and machine managers, an SMA holding $1,000 has the same operational structure, the same custody guarantees, and the same protocol cost as one holding $1,000,000. The marginal cost of personalization, in protocol terms, is zero.

The agent economy is the medium. The onchain SMA is the structure. Sail's mission is to unlock personalized finance for all.