ChatGPT

Consortium Distributed Ledger Architecture for Decentralized Identity (DID) in Go

Introduction

Decentralized Identity (DID) systems enable self-sovereign identity by allowing entities to manage their own identifiers and public keys without a centralized authority. To support DIDs across multiple industries, we propose a **consortium-based distributed ledger** as a Verifiable Data Registry (VDR) [1]. The consortium ledger will be **permissioned** (operated by a group of known organizations) but can allow public read access to identity data as needed. This architecture is designed for **scalability, security, and maintainability**, and uses **Go** as the primary implementation language. We will examine Ethereum-compatible and alternative ledger platforms with Go support – notably **Go-Ethereum/Quorum**, **Hyperledger Fabric**, and **Tendermint/Cosmos SDK** – and recommend a suitable technology stack and libraries in Go for building the DID registry. Key considerations include DID method support (e.g. **did:ethr**, **did:indy**, or custom methods), consensus and governance models for a consortium, and modules for features like revocation, credential anchoring, and audit logging.

Architecture Goals and Design Considerations

- **Multi-Industry Flexibility:** The ledger design should be generic and configurable to serve various industries (finance, healthcare, IoT, etc.), since the exact use cases are not yet finalized. This implies using open standards (W3C DID, Verifiable Credentials) and supporting multiple DID methods.
- **Consortium Governance and Security:** Only authorized consortium members (organizations) should run nodes and write to the ledger. Governance processes are needed for onboarding new members, defining endorsement policies for transactions, and upgrading network parameters. The design should enable shared trust (no single controlling party) through decentralized consensus and possibly on-chain governance mechanisms [2] [3].
- **Scalability and Performance:** The ledger must handle a high volume of DID operations (create/update/revoke) and credential anchoring transactions. Prior identity ledgers like Hyperledger Indy achieved only ~4 TPS due to expensive consensus [4], so the chosen platform should improve on this (tens to thousands of TPS). Solutions like Tendermint/Cosmos or optimized enterprise Ethereum clients have demonstrated significantly higher throughput (e.g., Cosmos-based **cheqd** network ~10,000 TPS [4]).
- **Maintainability and Extensibility:** Using Go allows leveraging mature open-source frameworks and libraries. The architecture should separate concerns (network layer, identity management logic, application layer) so that components can be maintained or replaced independently. Smart contracts or chaincode should encapsulate DID logic cleanly. Standard interfaces (DID resolvers, APIs) should be provided for application developers.
- **Interoperability:** Support standard DID methods or allow plugging in new methods. The system might support **did:ethr** (Ethereum DID Registry contract) [5], **did:indy** (Indy-style identity transactions on a ledger) [6], or a new consortium-specific method (e.g., `did:example:` for this ledger). Ensuring compliance with the W3C DID Core specification and resolution process is critical so that external DID resolvers and tools can interact with the ledger's DIDs.

- **Privacy and Data Minimization:** Only public keys, DID documents, and necessary metadata should go on-chain. Personal data from credentials should not be stored on the ledger; instead, only proofs or hash pointers are anchored. The architecture should accommodate *selective disclosure* and privacy-preserving revocation (e.g. cryptographic accumulators as used in Indy [7]).

High-Level Architecture Overview

【Architecture Diagram: Consortium DID Ledger】
(Diagram Description:) The DID platform consists of a **consortium blockchain network** (permissioned ledger nodes), a **DID Registry smart contract or chaincode** running on the ledger, and **client SDKs/agents** that interact with the ledger. Each consortium member (organization) runs one or more validator nodes participating in consensus. DID operations (create, update, deactivate) are submitted as transactions to the network, and upon consensus, the ledger's state is updated (e.g., storing a DID Document or an entry in a registry). Client applications (such as identity wallets, credential issuer services, or DID resolvers) use a Go SDK to submit transactions and query DIDs from the ledger.

Network Layer: The underlying blockchain provides **distributed ledger and consensus**. For a consortium, a *Byzantine Fault Tolerant* algorithm is typically used to tolerate malicious or failing nodes. Options include Istanbul BFT (in Quorum/Ethereum), Raft or PBFT (in Fabric), or Tendermint BFT (in Cosmos). All consortium validator nodes collectively validate and agree on transactions, ensuring integrity and consistency of the DID registry. The network is permissioned – only approved entities' nodes join the validator set [6]. Permissions may be managed by a Membership Service (in Fabric) or by on-chain permissioning contracts (in enterprise Ethereum clients) [8].

Identity Ledger Layer: The core identity logic resides in smart contracts or chaincode that implement the **DID Registry** and related functions. This layer defines how DIDs are registered and resolved: - *DID Registration:* When a new DID is created, the registering party submits a transaction. The ledger enforces that only authorized identities can register or update certain DIDs (e.g., you can only update a DID document if you are its controller, as proven by a signature). For example, the **ERC-1056 EthereumDIDRegistry** contract (did:ethr method) uses on-chain access control so that only the DID controller (or delegates) can perform updates [5]. In permissioned networks, additional endorsement rules can apply (e.g., requiring an organizational endorser signature for writes, similar to Indy's endorser model [3]).
- *DID Document Storage:* Depending on the method, either full DID Documents are stored on-chain or the chain stores references. A simple design is to store the DID Document JSON (or its components: public keys, authentication methods, service endpoints) as state in the ledger (e.g., as key-value with key = DID). This can be done via chaincode in Fabric or a smart contract in Ethereum. Alternatively, a **Sidetree-based** approach (used by *did:trustbloc*) stores only hashed operations on-chain and keeps full documents in off-chain storage, improving scalability [9]. The choice affects performance and complexity. For a generic solution, starting with on-chain DID Documents for simplicity is reasonable, with an option to migrate to Sidetree if needed for scale.
- *Query and Resolution:* The ledger exposes query APIs (or RPC) to retrieve DIDs and associated documents. In Go, one can implement a DID **Resolver** module that queries the blockchain state and constructs the DID Document as per the DID method spec. For example, resolving a did:ethr involves reading the EthereumDIDRegistry contract events to reconstruct the latest DID document [5]. In our architecture, a Go DID resolution service (possibly using libraries like `nuts-foundation/go-did` or Hyperledger Aries components) will allow applications to resolve DIDs by calling ledger APIs.

Application Layer: On top of the ledger, various identity services and applications will use the DIDs: - **Issuer/Verifier Services:** Entities that issue Verifiable Credentials or verify them will use the DID ledger

to fetch public keys (for signature verification) and possibly to publish credential schemas or status lists.
- **Identity Wallets and Agents:** User-side software (mobile wallets, agent servers) interacts with the ledger via a REST API or SDK to register their DID or update keys. These clients might incorporate Go libraries for DID communication (e.g., **Aries Framework Go** for DIDComm protocols, though now archived, or other DIDComm v2 libraries) to exchange credentials off-ledger.
- **Governance/Admin Tools:** There will be administrative applications for consortium governance (adding new member organizations, rotating keys of nodes, updating chaincode or smart contracts with consensus of members). These might leverage on-chain governance features if available (Cosmos SDK provides an on-chain voting module [2]), or use off-chain agreements reflected via multi-sig transactions (common in enterprise Ethereum or Fabric channel updates).

All interactions are secured with strong cryptography. Transactions to the ledger are signed by the sender's private key (in Fabric, X.509 certificates issued by the CA; in Ethereum/Cosmos, ECDSA keys with accounts), ensuring authenticity. DID Documents contain public keys and service endpoints, and the ledger's consensus and immutability guarantee that once a DID is registered/updated, the history is auditable and tamper-evident.

Comparing Distributed Ledger Frameworks (Go-Compatible)

Choosing the right ledger framework is crucial. The table below compares three leading options that support Go development and Ethereum compatibility:

Framework	Type & Consensus	Go Support	DID Method Support	Key Features for Consortium DIDs
Go-Ethereum / Quorum (Ethereum)	Ethereum-based EVM chain. Supports PoA consensus (e.g. IBFT 2.0, QBFT in Hyperledger Besu) for consortium [8]. Quorum (Go) is a fork of Geth for private networks.	**Strong Go support:** `go-ethereum` client in Go; Go SDKs (web3 libraries) for contract interaction. Smart contracts written in Solidity (not Go) but can be deployed and called from Go.	**did:ethr** – Ethereum DID Method via ERC-1056 registry [5]. Other EVM-based DID methods possible (did:jolo, did:dock, etc.) using custom contracts.	- **Enterprise features:** Besu/Quorum add permissioning and privacy (private transactions) suitable for consortia [10]. - **Broad adoption:** Ethereum tooling (e.g., MetaMask, Truffle) and existing DID libraries (uPort ethr-DID) available. - **Scalability:** IBFT can reach high TPS in permissioned settings; however, transaction finality is typically slower (~5s+) and careful gas management is needed. - **Interoperability:** Ethereum networks can interoperate with public chains if needed, and Besu is enterprise-friendly for both public and private use [8].

Framework	Type & Consensus	Go Support	DID Method Support	Key Features for Consortium DIDs
Hyperledger Fabric	Permissioned ledger with modular consensus (solo/raft ordering). Not EVM-based (uses chaincode model). Transactions validated via endorsement policy and ordered into blocks via ordering service.	**Native Go support:** Fabric is written in Go; chaincode (smart contracts) can be written in Go. Fabric SDK for Go is available for client apps.	**Custom DID Method:** No default DID method, but can implement e.g. `did:fab` (non-standard) or use as VDR for did:indy or others. Chaincode can be written to follow W3C DID operations [11].	- **Permissioned and Private:** Only member nodes see data, enabling privacy. Fine-grained access control via channels and endorsement policies. Ideal when data sharing must be restricted to certain parties [12] [13]. - **Performance:** Proven high throughput (thousands of TPS) in permissioned environments for simple transactions. Parallelism via channels and no global token economy. - **Governance:** Consortium governance handled through channel config updates and Membership Service Provider (MSP) for identities. No built-in token or on-chain voting, governance is more off-chain/administrative. - **Identity integration:** Fabric uses X.509 certificates for node identity, which can complement DID usage (e.g., tying an org's DID to its MSP identity). DID registry chaincodes have been implemented (e.g., by Oracle and others) to register and resolve DIDs on Fabric [11].

Framework	Type & Consensus	Go Support	DID Method Support	Key Features for Consortium DIDs
Tendermint/ Cosmos SDK	Cosmos SDK allows building a custom blockchain in Go. Tendermint BFT provides fast finality (1-2s) and high throughput. Cosmos networks are typically public/ permissionless by default, but validator set can be permissioned by consortium agreement.	**Excellent Go support:** Cosmos SDK is a Go framework. Developers write application-specific modules in Go. Many Cosmos-based networks (like **cheqd**) are built for SSI [14].	**Custom DID Method:** Cosmos-based networks can define their own DID method (e.g., `did:cheqd`). Modules for DID operations can be built or integrated. Cheqd's DID method is an example of Cosmos for DIDs [15]. Potential to also support did:indy style roles or interoperable methods.	- **Customization:** Full control to design identity-specific logic (e.g., on-ledger credential schemas, payment for DID operations). Cheqd network illustrates a Cosmos SSI ledger supporting W3C DIDs and even Indy's credential formats [16]. - **On-chain Governance:** Token-based governance (voting by validators/ token-holders) to upgrade network or add validators [2]. This can decentralize consortium decision-making (proposals, votes) if desired. - **Performance:** Tendermint BFT is high-performance; Cosmos chains can handle magnitudes more transactions than Indy (cheqd: 10k TPS vs Indy's 4 TPS) [4]. - **Trade-offs:** Requires more development effort to build and maintain a custom chain. A token economy is inherent (for fees and staking), which the consortium must manage (though fees can be minimized or handled internally).

Summary: All three approaches support a Go development stack but serve different needs. **Ethereum/ Quorum** offers compatibility with a widely-used DID method (did:ethr) and smart contract flexibility, making it easier to leverage existing tools. **Hyperledger Fabric** provides a mature enterprise-ready solution with Go chaincode, ideal for strict privacy and a token-less model, at the cost of more complex setup (CAs, orderer nodes) and slightly less standard DID integration (requires custom chaincode logic).

Cosmos SDK provides ultimate flexibility and performance, and has been successfully used in identity networks (e.g. cheqd), but entails building a new blockchain and managing tokens.

For a consortium aiming to be **industry-agnostic**, Fabric or a Cosmos-based network are attractive since they can be tailored to specific governance needs without exposing everything on a public chain. If **Ethereum compatibility** and existing DID method adoption are priorities, a permissioned Ethereum network using Quorum or Hyperledger Besu with an ERC-1056 DID registry smart contract is a strong option. In fact, efforts like *Hyperledger Indy on Besu* are combining Indy's identity features with Ethereum tech [17], reflecting the convergence of these approaches.

Go Technology Stack and Libraries for DID Systems

Leveraging Go, we can assemble a robust technology stack for implementing and interacting with the DID ledger:

- **Ledger Node Implementation:** If using **Go-Ethereum/Quorum**, the nodes run on the Go-Ethereum client. For **Fabric**, peer and orderer nodes from Hyperledger Fabric (Go binaries) will be deployed. For **Cosmos**, a custom blockchain binary (in Go) will be built using Cosmos SDK frameworks (by extending the `x/` modules or using Starport CLI to scaffold a new chain). All these are Go-based, ensuring the core platform can be maintained or extended in Go.

- **Consensus and Networking:** These are mostly built-in to the chosen platform (e.g., IBFT in Quorum, Raft/Etcd in Fabric's ordering service, Tendermint Core in Cosmos). Go developers typically do not implement consensus from scratch but configure it. For example, in a Cosmos chain, one might configure Tendermint parameters (block time, validator weights) via genesis settings. If needed, the **Tendermint RPC library (in Go)** can be used to monitor blocks and events. In Fabric, consensus is abstracted; Go devs configure the orderer (which uses Raft consensus under the hood).

- **Smart Contracts / Chaincode:**

- *Ethereum:* Write the DID Registry contract in Solidity (ERC-1056 standard can be adopted or a custom version). Go can be used to compile, deploy, and interact with this contract using **Go Ethereum's** `bind` **package** to generate Go contract APIs. E.g., use `abigen` to create a Go wrapper for the EthereumDIDRegistry contract, enabling Go code to call `CreateDID` or `UpdateDID` functions directly.
- *Hyperledger Fabric:* Implement chaincode in Go for DID operations. Fabric's chaincode SDK (the **shim API**) allows defining key-value state updates. For instance, a `RegisterDID(ctx, did, doc)` function in chaincode could put the DID Document JSON into state. Another function `ResolveDID(ctx, did)` could retrieve it. Endorsement policies can be set such that, say, *any two consortium orgs must endorse a DID registration*, or more simply any single org's peer can endorse its own DID writes. The **Fabric Go SDK** on the client side will invoke these chaincode functions from applications.

- *Cosmos SDK:* Develop a custom **DID Module** in Go. This involves defining the state structures (e.g., a `DIDDocument` struct), message types for transactions (MsgCreateDID, MsgUpdateDID, etc.), and handling them in module logic. Projects like *cheqd* have open-sourced their DID module which can be a reference [14]. The Cosmos module can also expose gRPC/REST endpoints that wallets or agents call.

- **DID Document & Credential Handling:** For parsing, validating, and constructing DID Documents and Verifiable Credentials in Go, libraries like `github.com/nuts-foundation/go-did` are very useful. This library can parse DID Documents and Verifiable Credentials (JSON-LD or JWT) and produce Go objects [18] [19]. It supports creating DIDs, generating keys, and serializing documents. Using such a library ensures our DID Documents comply with W3C standards and can be easily manipulated in code (adding verification methods, services, etc.). Another resource is **Hyperledger Aries Framework Go**, which (while recently archived) provides a comprehensive set of Go packages for DIDs, DIDComm, and credential exchange. Aries Framework Go includes a DID resolver interface and implementations for did:peer and did:indy, as well as utilities for packing/unpacking DIDComm messages. Early-stage developers can leverage Aries components if building higher-level identity agents that interact with the ledger. For lower-level DID operations and crypto, Aries provides the **Aries Crypto (Aries Crypto/URSA)** support for keys and signatures (including BLS, ED25519, etc.).

- **Identity SDKs and APIs:** We will expose functionality via a Go-based SDK to client applications. This might be a custom library that wraps ledger interactions (calls to Ethereum JSON-RPC or Fabric peer) and provides easy methods like `CreateDID(doc)`, `GetDIDDocument(did)`, `RevokeDID(did)`. In Ethereum context, one could use the official `ethclient` package (from go-ethereum) to connect to a node and the `bind` auto-generated Go contract stubs to call DID methods. In Fabric, the **Fabric SDK Go** allows submitting transaction proposals to peers and handling the endorsement and commit. In Cosmos, one can use the **Cosmos SDK's gRPC gateway** or the **tendermint RPC** for queries; additionally, Cosmos provides a Go client (via Protobuf-generated code for transactions).

- **Supporting Services:** For functions like **audit logging and analytics**, the stack can include an off-chain database or logging service that subscribes to ledger events. For example, a Go daemon could use **Fabric event listeners** or **Ethereum event subscriptions** (via WebSocket) to log all DID events (creations, updates) into an audit database, providing an easy search and compliance log. Similarly, a **block explorer** tailored to identity transactions could be built (there are open-source examples for Ethereum and Fabric explorers).

- **Crypto Libraries:** Go has strong crypto in its standard library and in packages like `crypto/ecdsa`, `crypto/ed25519` which will be used for key operations. If Hyperledger Ursa (for ZKP credentials like AnonCreds) is needed, wrappers or an RPC to a service might be used since Ursa is C++/Rust. However, for modern W3C credentials, we can use JSON-LD Signatures and BBS+ libraries. The **Aries BBS+ Go library** is available for BBS+ signatures if needed (for selective disclosure credentials) [20].

- **External Integrations:** The design can integrate with **Universal DID Resolver** infrastructure. For instance, the consortium can publish a driver such that a DID Resolver knows how to query our ledger for `did:example:corp:123` identifiers. This typically means providing a resolution endpoint or implementing a DID resolution interface in Go and deploying it as a service. This ensures interoperability – any user globally can resolve DIDs from this ledger using standard tools.

Consortium Governance, Endorsement and Registry Structure

A crucial aspect of a consortium DID ledger is how the **governance and trust model** is structured:

- **DID Registry Structure:** In a multi-organization environment, the ledger may enforce rules on who can write which DID records. One common pattern (seen in Hyperledger Indy's did:indy method) is to designate certain roles like *Trust Anchors* or *Endorsers* who have permission to write new DIDs [3]. We can adopt a similar approach: for example, require that a new DID created by an individual or device is countersigned by a sponsoring organization (a member of the consortium) to be accepted on-chain. This ensures each identity is anchored by a trusted member. Alternatively, if the DID owners themselves are the consortium members (e.g., enterprise DIDs), we might allow each org to write its own DIDs freely but not others'. Technical enforcement can be via chaincode logic or contract logic that checks the writer's identity against the DID namespace (e.g., OrgA can only create DIDs under `did:example:orga:*`). In Ethereum, this could be encoded as a requirement that certain transactions must be signed by specific keys or pass a multi-sig check. In Fabric, endorsement policies could require, say, "M out of N orgs endorse the transaction" before commit – providing a governance checkpoint for critical operations.

- **Endorsement Model:** Hyperledger Fabric uses an endorsement policy for transactions, meaning some set of peers must endorse (sign) a transaction proposal before it's accepted. For a DID registry chaincode, we could set an endorsement policy such that at least 2-of-3 (or majority) of different orgs endorse any DID creation. This gives a level of review – e.g., Org1's peer proposes a DID write, Org2 or Org3's peer must endorse it as well. This could prevent a single malicious actor from flooding the ledger with bogus DIDs. In Ethereum-based networks, endorsement is not baked in, but one can achieve similar outcomes by multi-signature schemes or requiring consortium consensus at the governance layer (outside the chain). Because our ledger is BFT, every transaction ultimately goes through consensus of validators; this itself provides security that invalid transactions (not following rules) are rejected. Additionally, *did:trustbloc* introduced a concept of *consortium endorsement* for configuration: members sign a consortium config file which lists the ledger's genesis and endpoints [21]. This ensures everyone agrees on the network's composition. We recommend establishing a **governance policy document** that all members sign (out-of-band) to define roles, permissions, and fees – similar to Sovrin's governance for Indy [22] – and then enforce parts of it in the ledger logic (through access controls and endorsement checks coded in smart contracts/chaincode).

- **Governance Model:** Consortium governance can be handled off-chain by a governance board, or on-chain through voting mechanisms:

- *Off-Chain Governance:* The consortium could form a steering committee that decides on new members, network parameter changes, or DID method updates. Decisions are documented and then executed on the technology (e.g., adding a new member's node certificate to the network config in Fabric, or updating an allowlist contract in Ethereum). This model was used by Hyperledger Indy networks – governance policies are agreed offline and validators sign legal agreements [22]. It's a straightforward approach for a permissioned network, especially in regulated industries where formal contracts are needed.
- *On-Chain Governance:* Emulating Cosmos governance, we could allow consortium members to stake votes on proposals recorded in a governance contract or module. For example, a smart contract could accept proposals (like "Add Organization X as validator") and members' votes (with their identities or tokens). If a proposal passes, the contract could trigger logic (if supported) or

at least record the decision which consortium operators then implement. Cosmos SDK has this built-in (validators vote with stake) [2] ; in Fabric/Ethereum, one might implement a simple voting contract or just use multi-sig approval for changes.

The governance model also ties into **economic model**: Will the network charge fees for DID operations? Public networks like Sovrin charged transaction fees to write DIDs [3] . A consortium might decide to waive fees or use an internal token for rate-limiting. If using Cosmos-based chain, a native token exists (like CHEQ on cheqd) which could be used to prevent spam and enable token incentives [23] . In Fabric, there is no native token, so either transactions are "free" or the consortium can use an external billing system if needed. Since the question focus is more on architecture, we note that this is a governance choice: a **fee-less permissioned ledger** is simplest (members absorb the costs of running nodes), but if scalability to a broader network or sustainability is a concern, introducing a token or fee for writes can be considered.

- **Data Models for DIDs:** The on-chain data model should be designed for **efficiency and compliance**. A DID Document can be stored as a single JSON string, but updating a single key would then rewrite the whole document. An alternative is to store each DID's components (authentication keys, services, etc.) as separate state entries (this is how the Ethereum DID Registry works via events/attributes [5]). For example, ERC-1056 logs events for adding a key, revoking a key, setting an attribute, etc., and the *did:ethr resolver* client reconstructs the DID Document by aggregating those events [24] [5] . This event-sourced approach is efficient on Ethereum. In Fabric or Cosmos, we could either mimic that (store a list of changes) or maintain a canonical state object per DID that gets overwritten. Given a permissioned context with likely fewer cost constraints, maintaining an explicit state object (e.g., a struct with DID, controller, list of Verification Methods, list of Services, etc.) might be simplest, and each update transaction replaces it (with history still available in the ledger's transaction log for audit).

- **Consortium DID Method:** If the consortium creates its own DID method (say `did:consortium:`), it should publish a DID Method specification describing how to perform CRUD operations on the ledger for that DID. For instance, the method spec would detail that to create a DID, one must submit a transaction to this specific ledger and what constitutes a DID Document on this ledger. It will also specify the DID URI format (perhaps including a namespace for the network, like `did:consortium:ledger1:abc123`). This ensures interoperability – others can integrate support for our DIDs. We can follow existing method specs as templates (did:indy's spec defines similar operations on an Indy ledger [25] [26] , did:trustbloc's spec defines how multiple orgs run a Sidetree DID registry [9]).

In summary, the registry and governance design should strike a balance between decentralization and control. Each consortium member should have a say (no single party unilaterally controls identities), but governance processes should not overly hinder legitimate identity operations. A layered trust model (members trust each other to endorse and validate, end-users trust the consortium to maintain the ledger integrity) will underpin the system's credibility.

Optional Modules and Features

In addition to core DID operations, the architecture can include modules to handle **revocation, credential status, and audit logging** to support full lifecycle of decentralized identities:

- **Revocation Mechanisms:** Revocation can refer to *DID deactivation* or *credential revocation*. DID deactivation is typically an update to the DID Document marking it as deactivated (or removing all keys). The ledger should allow an authorized party (DID controller or consortium admin in

some cases) to mark a DID as deactivated so resolvers know it's no longer valid. For credential revocation, a common pattern is to maintain a **revocation registry** on the ledger. Hyperledger Indy implemented revocation by storing cryptographic accumulators on-ledger; each accumulator update (revocation entry) marks certain credentials as revoked in a privacy-preserving way [27]. In a more generic DID system, we could use a simpler approach: for example, maintain a list or bitmap of revoked credential IDs. One emerging W3C standard is the **Status List 2021** approach (which uses a bitstring to denote revoked credentials in a list, often stored as a JSON in a public URL). Our ledger could store the hash or URI of such a status list that issuers update. Alternatively, each credential's status could be an entry on-chain (suitable if volume is low). The architecture can include a **Revocation Manager** chaincode/contract that issuers use to post revocation events. For instance, an issuer could submit a transaction, "Revoke credential X of DID Y", which the ledger records. Verifiers checking that credential would query the ledger for any revocation entry. This module should be optimized for lookup (perhaps indexing by credential ID or by issuer). In a consortium, revocation entries could be endorsed by the issuer's organization to prevent false revocations.

- **Credential Anchoring:** In many DID systems, credentials themselves are not stored on the blockchain, but a hash or fingerprint of the credential can be anchored on-chain to provide tamper evidence and timestamping. Our architecture can support an **anchoring service** where, whenever a Verifiable Credential is issued, the issuer (or the holder) submits a transaction containing a cryptographic digest of the credential (e.g., a SHA-256 hash). This creates an immutable timestamped record proving that a credential with that hash was created at that time. To protect privacy, only the hash is stored – the actual credential is shared off-chain between holder and issuer. Later, a verifier can hash the credential presented to them and check the ledger to see if a matching hash exists (and optionally if it's not revoked). This mechanism is lightweight and can handle cross-industry use cases (for example, anchoring academic certificates, KYC credentials, etc.). It's similar to the approach used by some networks like Alastria, which anchors *credential transactions via multi-hash* for privacy [28]. We can implement credential anchoring as a simple chaincode that maps `credentialHash -> issuer DID or status`. This could also be extended to anchor **presentations** or audits (though that may be beyond initial scope).

- *Integration with DIDs:* A DID Document could include a service endpoint referring to a credential status list or registry on the ledger. For example, a DID Document might have a `service` entry of type `RevocationList2020` that points to an on-chain resource (or off-chain URL whose hash is on-chain) listing revoked credentials issued by that DID. This ties credentials to the DID method design.

- **Audit Logging and Monitoring:** Every operation on the ledger is recorded in its blockchain log, providing an immutable audit trail. We recommend building an **audit and analytics layer** on top of this to make it easy for administrators and possibly regulators to review identity events. In practice, this could be a combination of:

- A **block explorer** interface that filters transactions to show only DID-related events in a human-readable form (e.g., "DID X created by OrgA on 2025-07-02", "Public key updated for DID Y by OrgB on 2025-08-10"). This can be adapted from existing blockchain explorer tools, using the specific smart contract or chaincode data structures to decode the transactions.
- **Logging hooks**: If using Fabric, one can write chaincode that also emits events (Fabric chaincode events) on each DID operation. A Go event listener can catch these and store them in a secure log (e.g., an append-only file or database). Similarly, an Ethereum contract can emit events (which are inherently logged and can be listened to via Web3 subscriptions). These logs can be

fed into a SIEM system for monitoring unusual activities (e.g., an alert if a large number of DIDs are deactivated in short time, which might indicate a security issue).

- **Compliance and Reporting:** For multi-industry usage, different sectors might have compliance needs (e.g., GDPR – right to be forgotten, etc.). While blockchain data is immutable, we can design the DID data to avoid personal data on-chain, and any personal data stored off-chain can be erased as needed. Audit tools can help demonstrate compliance by showing exactly what data is on-chain for a given DID. Furthermore, if the consortium needs to support audits by external parties, an **audit API** could provide read-only access to ledger data with cryptographic proofs (like proving a certain DID state at a certain block via a Merkle proof). Libraries like **go-merkle** can assist in generating proofs from the state trie (for Ethereum) or state database (Fabric has queryable state with history).

By incorporating these modules, the DID system becomes more robust and feature-complete: issuers can manage credential lifecycles, verifiers have up-to-date trust information, and administrators have oversight. Each module can be enabled or disabled based on the use-case (for example, if initial use-cases don't require credential revocation, that module can be deferred).

Conclusion and Recommendations

Designing a consortium-based DID ledger in Go is feasible with today's technology stack, and it offers a powerful foundation for cross-industry decentralized identity. We have outlined a scalable and secure architecture that leverages a permissioned blockchain as the verifiable data registry [1], supporting standard DID methods and optional identity services.

For the **technology platform**, we recommend evaluating **Hyperledger Fabric and Cosmos SDK** as the top choices given their native Go support and flexibility. **Hyperledger Fabric** is ideal if your consortium prioritizes fine-grained data privacy and wants to use familiar enterprise tools; it allows implementing DID logic via chaincode and has been used in many industries [13]. **Tendermint/Cosmos SDK**, on the other hand, offers excellent performance and modularity – if you anticipate a global utility network or need on-chain governance with a token economy, a Cosmos-based identity network (like the example of cheqd) could be the future-proof path. If Ethereum compatibility and existing DID method adoption (did:ethr) is more important, a private Ethereum network (Go-Ethereum/Quorum or Hyperledger Besu with Go client bindings) can be chosen, deploying the ERC-1056 DID Registry contract and related identity smart contracts. This Ethereum-based approach benefits from widespread tooling and a ready-made DID method [5], though it introduces Solidity into the stack for smart contracts.

In all cases, a rich set of Go libraries and SDKs will accelerate development: - Use **go-ethereum** or **Fabric SDK Go** to interface with the ledger. - Leverage identity libraries like **go-did** for DID Document handling and **Aries Framework Go** (or its components) for higher-level protocols. - Implement robust governance with multi-signature contracts or Fabric policies, ensuring only authorized writes and transparent consortium decision-making. - Incorporate revocation and credential status tracking as first-class features to support full SSI workflows, learning from Indy's and W3C's approaches [27]. - Plan for interoperability by documenting the DID method (so that, for example, the W3C DID Directory could list your method alongside did:indy, did:ethr, etc.) and possibly bridging to other networks if needed (via DID hubs or cross-ledger resolvers).

Maintainability is addressed by sticking to Go across the stack – consortium members can use a common language for chain code, server-side agents, and client libraries, easing the DevOps and knowledge sharing. **Security** is ensured by using battle-tested blockchain protocols (BFT consensus, audited smart contract code for DID registry) and by implementing strict access controls (both on-chain and at the network layer). As the consortium grows or the specific industry requirements solidify, this

architecture can adapt – new modules can be introduced (e.g., integrating zero-knowledge proofs for private credential proofs, or adding support for additional DID methods required by certain regulators) without fundamental redesign.

In summary, the consortium DID ledger architecture presented here is a modular **technical architecture brief** for early-stage developers and architects. By following this design and using the recommended tools and libraries, one can build a decentralized identity platform in Go that is scalable across industries, compliant with emerging standards, and governed by the consortium that uses it. This will lay a strong foundation for any SSI (Self-Sovereign Identity) solutions that the consortium's members want to deploy, providing the trust backbone (the blockchain VDR) for years to come.

Sources: The design and recommendations are informed by industry standards and projects in decentralized identity, including W3C DID specifications, Hyperledger open-source projects, and real-world networks like Sovrin/Indy, Ethereum DID Registry, and Cosmos-based identity networks like cheqd. Key references have been provided throughout [6] [5] [4] [9] to substantiate the architectural choices.

---

[1] [2] [3] [6] [7] [22] [25] [26] [27] How Do Blockchains Provide the Trust Foundation for Decentralized Identity-Based Apps?
https://anonyome.com/resources/blog/how-do-blockchains-provide-the-trust-foundation-for-decentralized-identity-based-apps/

[4] [16] [23] Deep-dive in our portfolio company: Cheqd.io | by Blue Node Capital | Medium
https://bluenodecapital.medium.com/deep-dive-in-our-portfolio-company-cheqd-io-6aeb11a2a603

[5] [24] Decentralized Identifiers: What and How?
https://tatum.io/blog/decentralized-identifiers-dids

[8] [17] Besu
https://www.lfdecentralizedtrust.org/projects/besu

[9] [21] [28] DID Methods - Various | Verifiable Credentials and Self Sovereign Identity Web Directory
https://decentralized-id.com/web-standards/w3c/decentralized-identifier/did-methods/

[10] [12] [13] 4 most popular blockchains -analysis and comparison of Ethereum, Hyperledger Fabric, Corda and Quorum - Nextrope - Your Trusted Partner for Blockchain Development and Advisory Services
https://nextrope.com/4-most-popular-blockchains-analysis-and-comparison-of-ethereum-hyperledger-fabric-corda-and-quorum/

[11] Enabling Decentralized/Self-Sovereign Identity with ... - Oracle Blogs
https://blogs.oracle.com/blockchain/post/enabling-decentralizedselfsovereign-identity-with-oracle-blockchain-platform

[14] Ledger/node software for cheqd's decentralised identity network ...
https://github.com/cheqd/cheqd-node

[15] ADR 001: cheqd DID Method - Product Docs
https://docs.cheqd.io/product/architecture/adr-list/adr-001-cheqd-did-method

[18] [19] GitHub - nuts-foundation/go-did: Golang library for parsing Decentralized Identifiers (DIDs)
https://github.com/nuts-foundation/go-did

[20] hyperledger/aries-bbs-go - GitHub
https://github.com/hyperledger/aries-bbs-go