Solana Protocol Design for Agent and MCP Server Registries

1. Introduction

Purpose and Motivation

The proliferation of autonomous agents and sophisticated AI model integrations signifies a paradigm shift in digital ecosystems. These intelligent entities, capable of independent action and complex decision-making, promise to revolutionize various domains, from decentralized finance (DeFi) to supply chain management and beyond.[1] However, as the number and diversity of these agents and AI-driven services grow, a fundamental challenge emerges: their effective discovery, verification, and interoperability within decentralized networks. Without standardized registries, locating reliable and capable agents or Model Context Protocol (MCP) servers becomes a fragmented and inefficient process, hindering the formation of a cohesive and dynamic AI ecosystem.

This report addresses this challenge by proposing two interconnected Solana-based program protocols: an Agent Registry and an MCP Server Registry. These on-chain registries are designed to serve as foundational infrastructure, enabling robust discoverability, fostering trust through verifiable information, and promoting interoperability among diverse AI components. By leveraging the high-performance characteristics of the Solana blockchain, these protocols aim to provide a scalable and efficient solution for managing and discovering AI services.

Scope of the Report

This document provides a detailed technical specification for the Solana program protocols governing the Agent Registry and the MCP Server Registry. The primary focus is on the on-chain data structures, the mechanisms for registration, update, and deregistration of entities, and advanced strategies for discoverability, search, and querying. The design incorporates insights from established agent frameworks such as the Autonomous Economic Agent (AEA) framework [1] and Google's Agent-to-Agent (A2A) protocol [3], as well as the Model Context Protocol (MCP) specification.[5]

Key Objectives

The principal objectives of this protocol design are:

1. To define clear, precise, and implementable on-chain data specifications for both Agent and MCP Server registry entries, ensuring data integrity and efficient storage.
2. To outline secure and gas-efficient Solana program instructions for the lifecycle

management of registry entries, including registration, updates, and deregistrations.
3. To propose a multi-layered approach to discoverability, combining direct on-chain lookups with powerful off-chain indexing and querying capabilities facilitated by on-chain event emission.
4. To ensure the Agent Registry design reflects the functional and economic characteristics of AEAs and the discovery paradigms of A2A.
5. To align the MCP Server Registry with the official MCP specification, particularly concerning the advertisement of tools, resources, and prompts.

2. Foundational Solana Concepts for On-Chain Registries

The design of on-chain registries on the Solana blockchain is fundamentally shaped by its unique architecture, including its account model, data storage mechanisms, and transaction processing. Understanding these core concepts is crucial for developing efficient, scalable, and secure registry protocols.

Program Derived Addresses (PDAs) for Registry Entries

Program Derived Addresses (PDAs) are a cornerstone of Solana program development, offering a mechanism to create accounts that are controlled by a program rather than a private key.[7] A PDA is deterministically derived from a program's ID and a set of "seeds"—byte arrays defined by the developer.[9] These seeds can include unique identifiers, categories, or other metadata, allowing for the creation of unique addresses for each registry entry. For instance, an agent's unique ID could serve as a seed to derive its specific PDA within the Agent Registry. Solana allows up to 16 seeds, each with a maximum length of 32 bytes, to derive a PDA.[7]

The use of PDAs is integral to the proposed registries for several reasons. Firstly, they provide a tamper-proof, uniquely addressable storage location (an account) for each registered agent or MCP server. Secondly, since PDAs are program-controlled, the registry program itself can authorize modifications to the data stored within these PDA accounts, ensuring that only legitimate actions (e.g., updates by the registered owner) are permitted. The selection of seeds for PDA derivation directly influences basic on-chain lookup capabilities; if a unique identifier is part of the seeds, direct retrieval of a specific registry entry becomes highly efficient, analogous to a key-lookup in a key-value store.[8]

Account Structures and Data Serialization (Borsh)

In Solana, all data, including program code and state, is stored in accounts.[7] Each account has an address (its public key or a PDA), stores data in a byte array, holds a

balance of lamports (Solana's native token unit) to cover rent, is owned by a specific program, and has flags indicating if it's executable or writable.[7] The data for each Agent Registry and MCP Server Registry entry will be stored within the data field of its respective PDA account.

For serializing this data into a compact binary format suitable for on-chain storage, Borsh (Binary Object Representation Serializer for Hashing) is the mandated standard.[12] Borsh is favored for its efficiency, determinism, and security-critical focus, providing a well-defined schema for data organization.[12] All on-chain data structures defined for the registries must be Borsh-compatible. This implies that schemas must be meticulously defined upfront. While Borsh's efficiency is a significant advantage, its fixed-format nature means that querying arbitrary fields within the serialized data blob directly on-chain is not feasible; the data must be deserialized to be interpreted, a process too computationally expensive to perform across numerous accounts during an on-chain query. This constraint underscores the importance of strategic PDA seed design for primary lookups and robust event emission for enabling sophisticated off-chain querying.

Rent, Storage Considerations, and Account Lifecycle

Storing data on Solana incurs "rent," a mechanism to account for the cost of maintaining data on the validators' storage.[11] To avoid ongoing rent payments, accounts must be made "rent-exempt" by maintaining a lamport balance equivalent to at least two years' worth of rent fees.[10] The amount of rent required is proportional to the size of the data stored in the account.[11]

Solana imposes a maximum storage size of 10MB per account.[7] While generous, this limit has significant implications for registry design. Storing extensive metadata, such as detailed descriptions of numerous agent skills or complex MCP tool schemas, directly within a single PDA could approach or exceed this limit, or at least make accounts very expensive to create and manage due to the upfront rent-exemption deposit. This consideration naturally leads to a hybrid storage model: core, verifiable information is stored on-chain, while more extensive or less frequently accessed details can be linked via URIs to off-chain storage solutions like IPFS or Arweave. This is analogous to the logoURI in the Solana Token List or the uri field in Metaplex token metadata, which points to off-chain JSON metadata.[15] The lifecycle of a registry entry PDA will involve its creation (funded by the registrant to cover rent-exemption) and potential closure (allowing the registrant to reclaim the lamports if the entry is deregistered and the account closed).

On-chain Data Limitations and Indexing Approaches

Solana's architecture is optimized for transaction processing speed and throughput, not for complex database-like queries on arbitrary data fields.[16] Direct on-chain querying capabilities are largely limited to fetching an account's data if its address (PDA) is known or can be derived. While getProgramAccounts can retrieve all accounts owned by a specific program, filtering these accounts based on their internal data content client-side is inefficient for large datasets.

To facilitate more advanced on-chain querying, secondary indexing patterns can be implemented. This typically involves creating additional PDAs that map attribute values to the primary keys (PDAs) of the main registry entries.[9] For example, a PDA seeded with a specific tag could point to agents possessing that tag. However, managing these on-chain secondary indexes—especially for attributes with high cardinality or when an index entry needs to store a list of other PDAs (e.g., Vec<Pubkey>)—introduces significant complexity, computational overhead during updates, and can itself run into account size limitations.[17]

Given these on-chain limitations, a robust and scalable discovery strategy must be hybrid. Basic, direct lookups can be performed on-chain. For complex, multi-faceted search and querying (e.g., "find all active agents with skill X that support protocol Y"), off-chain indexers play a critical role.[16] These off-chain systems listen to events emitted by the on-chain registry programs whenever an entry is created, updated, or deleted.[21] By processing these events, indexers can build and maintain sophisticated, queryable databases (e.g., SQL, Elasticsearch) that mirror and augment the on-chain data, providing the flexible search capabilities users expect.

3. Agent Registry Protocol Design

Core Philosophy

The Agent Registry protocol aims to establish a decentralized directory for autonomous agents operating on and interacting with the Solana ecosystem and beyond. It emphasizes the clear advertisement of agent capabilities, operational endpoints, identity, and relevant metadata to facilitate discovery and interaction. The design draws inspiration from the structured metadata approach of Google's A2A AgentCard [3], which provides a comprehensive schema for describing agent attributes, and the principles of Autonomous Economic Agents (AEAs), which highlight agents as independent economic actors with specific goals and capabilities.[1] The registry seeks to balance the richness of these specifications with the constraints and efficiencies of on-chain storage.

Data Specification for Agent Entry PDA (AgentRegistryEntryV1)

Each registered agent will have its data stored in a dedicated Program Derived Address (PDA) account. The structure of this data, AgentRegistryEntryV1, is defined to capture essential information for discovery, verification, and interaction, while also providing hooks for more extensive off-chain metadata.

The A2A AgentCard specification [3] is comprehensive, detailing fields for agent identity, capabilities, skills, service endpoints, and security requirements. Storing such an extensive structure entirely on-chain for numerous agents would be impractical due to Solana's account size limits (10MB [7]) and rent costs.[11] Similarly, AEA concepts introduce additional descriptive data points regarding economic intent and supported protocols.[1]

Consequently, the AgentRegistryEntryV1 structure adopts a hybrid approach:

1. **Core On-Chain Data:** Essential, frequently queried, and verifiable information is stored directly on-chain. This includes identifiers, ownership details, primary service endpoints, key capability flags, a summary of skills, status, and timestamps.
2. **Summaries and Hashes:** For potentially large or complex data elements (e.g., detailed skill descriptions, comprehensive security schemes), a hash of the data is stored on-chain for integrity verification, while the full data resides off-chain.
3. **Limited Collections:** For repeatable elements like tags or core skills intended for on-chain indexing or immediate display, fixed-size arrays or Vecs with clearly defined maximum lengths are used to manage storage.
4. **Off-Chain Extension:** An extended_metadata_uri field provides a link (e.g., to an IPFS or Arweave hosted JSON file) to a more comprehensive metadata document, which could be a full A2A AgentCard or a detailed AEA component manifest. This aligns with patterns like the A2A supportsAuthenticatedExtendedCard concept [3] and the uri field in Metaplex token metadata.[15]

This strategy optimizes on-chain storage costs and access performance while enabling rich, detailed, and verifiable agent descriptions.

The following table details the proposed AgentRegistryEntryV1 structure:

Table 1: Agent Registry Entry Data Structure (AgentRegistryEntryV1)

Field Name	Solana	Borsh	Description	AEA/A2A	Constraints

	(Rust) Type	Serialization Notes (Illustrative)		Mapping (if applicable)	/Examples
bump	u8	Single byte	The bump seed used for this PDA's derivation.	N/A	Required for PDA canonicalization. [9]
registry_version	u8	Single byte	Schema version of this entry (e.g., 1).	N/A	For upgradability.
owner_authority	Pubkey	32 bytes	Solana public key of the entry's owner/manager.	N/A	Critical for access control.
agent_id	String	Length-prefixed Vec<u8>	Unique identifier for the agent.	A2A: Implicitly, the agent's identity. AEA: Agent's unique ID.	Max 64 chars.
name	String	Length-prefixed Vec<u8>	Human-readable name of the agent.	A2A: AgentCard.name [3]	Max 128 chars.
description	String	Length-prefixed Vec<u8>	Human-readable description. CommonMark MAY be used.	A2A: AgentCard.description [3]	Max 512 chars.
agent_version	String	Length-prefixed Vec<u8>	Version of the agent software/implementation.	A2A: AgentCard.version [3]	Max 32 chars.

provider_name	Option<String>	Optional, then Length-prefixed Vec<u8>	Name of the agent's provider organization.	A2A: AgentCard.provider.organization [3]	Max 128 chars.
provider_url	Option<String>	Optional, then Length-prefixed Vec<u8>	URL of the agent's provider.	A2A: AgentCard.provider.url [3]	Max 256 chars, HTTPS recommended.
documentation_url	Option<String>	Optional, then Length-prefixed Vec<u8>	URL to human-readable documentation.	A2A: AgentCard.documentationUrl [3]	Max 256 chars, HTTPS recommended.
service_endpoints	Vec<ServiceEndpoint>	u32 count, then each struct	List of service endpoints.	A2A: AgentCard.url (generalized) [3]	Max 3 endpoints on-chain.
↳ protocol	String	Length-prefixed Vec<u8>	Protocol type (e.g., "a2a_http_jsonrpc", "aea_p2p").	N/A	Max 64 chars.
↳ url	String	Length-prefixed Vec<u8>	Endpoint URL.	N/A	Max 256 chars.
↳ is_default	bool	Single byte (0 or 1)	Indicates if this is the primary endpoint.	N/A	Only one can be true.
capabilities_flags	u64	8 bytes	Bitmask for core A2A capabilities.	A2A: AgentCard.capabilities (e.g., streaming, pushNotifs) [3]	e.g., 0x1=Streaming, 0x2=Push.

supported_input_modes	Vec<String>	u32 count, then each length-prefixed Vec<u8>	Default accepted input MIME types.	A2A: AgentCard.defaultInputModes [3]	Max 5 items, each max 64 chars.
supported_output_modes	Vec<String>	u32 count, then each length-prefixed Vec<u8>	Default produced output MIME types.	A2A: AgentCard.defaultOutputModes [3]	Max 5 items, each max 64 chars.
skills	Vec<AgentSkill>	u32 count, then each struct	Summary of key agent skills.	A2A: AgentCard.skills [3]	Max 10 skills on-chain.
↳ id	String	Length-prefixed Vec<u8>	Skill's unique ID within the agent.	A2A: AgentSkill.id	Max 64 chars.
↳ name	String	Length-prefixed Vec<u8>	Human-readable skill name.	A2A: AgentSkill.name	Max 128 chars.
↳ description_hash	Option<[u8; 32]>	Optional, then 32 bytes	SHA256 hash of detailed skill description (full description off-chain).	A2A: AgentSkill.description (hashed)	
↳ tags	Vec<String>	u32 count, then each length-prefixed Vec<u8>	Tags associated with the skill.	A2A: AgentSkill.tags	Max 5 tags, each max 32 chars.
security_info_uri	Option<String>	Optional, then Length-prefixed Vec<u8>	URI to detailed security scheme definitions (e.g., OpenAPI format).	A2A: AgentCard.securitySchemes, security (referenced) [3]	Max 256 chars.

aea_address	Option<String>	Optional, then Length-prefixed Vec<u8>	Fetch.ai AEA address/ID, if applicable.	AEA: Agent's address in AEA ecosystem. [25]	Max 128 chars.
economic_intent_summary	Option<String>	Optional, then Length-prefixed Vec<u8>	Brief summary of the agent's economic goals.	AEA: Economic focus [1]	Max 256 chars.
supported_aea_protocols_hash	Option<[u8; 32]>	Optional, then 32 bytes	SHA256 hash of a list of supported AEA protocol IDs (full list off-chain).	AEA: Protocols [2]	
status	u8	Single byte	Agent status (0:Pending, 1:Active, 2:Inactive, 3:Deprecated).	N/A	
registration_timestamp	i64	8 bytes (Unix timestamp)	Timestamp of initial registration.	N/A	
last_update_timestamp	i64	8 bytes (Unix timestamp)	Timestamp of the last update.	N/A	
extended_metadata_uri	Option<String>	Optional, then Length-prefixed Vec<u8>	URI to extensive off-chain metadata (e.g., full AgentCard JSON).	A2A: supportsAuthenticatedExtendedCard concept [3]	Max 256 chars (e.g., IPFS, Arweave).
tags	Vec<String>	u32 count, then each	General discoverabilit	N/A	Max 10 tags, each max 32

		length-prefixed Vec<u8>	y tags for the agent.		chars.

Registration and Management Instructions (Solana Program Functions)

The Agent Registry Solana program will expose several instructions to manage the lifecycle of agent entries. Access control is paramount: only the owner_authority specified in an agent's entry (and thus, the signer of the transaction who can prove ownership of that key) should be able to modify or delete it. This is enforced within each instruction by verifying the signer's public key against the stored owner_authority.

- **register_agent(ctx, agent_id: String, name: String, description: String,..., owner_authority: Pubkey, extended_metadata_uri: Option<String>)**:
  - This instruction initializes a new PDA account for the agent. The PDA is derived using seeds like ``.
  - The instruction populates the PDA's data account with the provided agent details, serializing them using Borsh.
  - The payer of the transaction (typically the owner_authority or an entity acting on their behalf) funds the new PDA account with enough lamports to make it rent-exempt.
  - Emits an AgentRegistered event containing the full AgentRegistryEntryV1 data.
- **update_agent_details(ctx, agent_id: String, new_name: Option<String>, new_description: Option<String>,...)**:
  - Allows the owner_authority to modify mutable fields of an existing agent entry.
  - The instruction fetches the agent's PDA, deserializes its data, updates the specified fields, and re-serializes the data.
  - Checks ensure the transaction signer is the owner_authority.
  - Emits an AgentUpdated event, detailing the agent_id and the fields that were changed along with their new values.
- **update_agent_status(ctx, agent_id: String, new_status: u8)**:
  - A specialized instruction for changing the agent's operational status (e.g., from Active to Inactive).
  - Requires owner_authority signature.
  - Emits an AgentStatusChanged event with agent_id and the new_status.
- **deregister_agent(ctx, agent_id: String)**:
  - Allows the owner_authority to remove an agent from active registration.
  - This could either mark the entry as "Deregistered" (preserving its history) or fully close the PDA account, reclaiming the lamports used for rent-exemption.

The choice depends on policy (whether to keep a permanent record of all registered agents, even defunct ones). If closing, care must be taken with associated secondary index entries.
    - Emits an AgentDeregistered event with agent_id.

Discoverability and Querying Mechanisms (Agent Registry)

Effective discovery is crucial for the utility of the Agent Registry. A multi-layered approach is proposed:

- **Direct Lookup by agent_id**: If the agent_id is known and used as a primary seed in PDA derivation (e.g., seeds = ["agent_v1", agent_id_bytes]), a specific agent's entry can be fetched directly and efficiently using solana_program::pubkey::Pubkey::find_program_address client-side, or by passing the derived PDA to an RPC call. This provides O(1) lookup complexity.
- **On-Chain Filtering (Limited)**:
    - Solana's getProgramAccounts RPC method can fetch all accounts owned by the registry program. Client-side filtering can then be applied by deserializing the data from each account. This is inefficient for large registries.
    - A more targeted on-chain approach involves incorporating highly selective, low-cardinality attributes into the PDA seed structure. For example, if status (e.g., 1 byte for 'Active') is part of the seeds (e.g., seeds = ["agent_v1", status_byte, agent_id_bytes]), getProgramAccounts could use a data filter on the prefix of the account data (if seeds are stored at the beginning) or more advanced RPC filters if available for seed components. However, PDA seeds have length limitations (max 16 seeds, each up to 32 bytes [7]), making it challenging to include many filterable attributes directly in seeds.
- **Event Emission for Off-Chain Indexing**: This is the most powerful mechanism for advanced search and querying. The registry program will emit detailed events for every significant lifecycle action:
    - AgentRegistered: Contains the full AgentRegistryEntryV1 data, including the agent_id, all on-chain fields, and the extended_metadata_uri.
    - AgentUpdated: Includes the agent_id and a representation of the changed fields (e.g., field name and new value, or the full updated entry).
    - AgentStatusChanged: Contains agent_id and the new status.
    - AgentDeregistered: Contains agent_id.

These events serve as a data feed for off-chain indexers.[16] These indexers can listen to the Solana chain, capture these events, fetch any linked off-chain data from extended_metadata_uri (e.g., the full AgentCard JSON), and populate a dedicated, query-optimized database (e.g., PostgreSQL, Elasticsearch). This off-chain database can then support rich, multi-faceted queries (e.g., "find all active agents with

'translation' skill, supporting 'a2a_http_jsonrpc' protocol, and tagged 'finance'"). Anchor's emit_cpi! macro is recommended for emitting critical event data, as it uses a Cross-Program Invocation to the program itself to log event data within instruction data, making it less susceptible to log truncation by RPC providers compared to the standard emit! macro, albeit at a higher Compute Unit (CU) cost.[21] Solana logs are primarily for real-time information transfer rather than historical querying like Ethereum events.[22]

The design of these events is as critical as the on-chain data structures themselves, as they form the primary interface for sophisticated off-chain discovery services.

4. MCP Server Registry Protocol Design

Core Philosophy

The MCP Server Registry is designed to facilitate the discovery of Model Context Protocol (MCP) compliant servers within the Solana ecosystem and beyond. MCP standardizes how AI applications (clients) interact with external data sources and tools (servers) by defining communication over JSON-RPC 2.0.[6] The registry will allow MCP server providers to advertise their services, including the Tools, Resources, and Prompts they offer, as outlined in the MCP specification.[6] The goal is to create a central, verifiable directory that enables AI applications to dynamically find and integrate with relevant MCP servers.

Data Specification for MCP Server Entry PDA (McpServerRegistryEntryV1)

Similar to the Agent Registry, each MCP server will be represented by a unique PDA account storing its metadata. The McpServerRegistryEntryV1 structure must capture the essence of an MCP server's identity and capabilities as defined in the MCP specification, particularly the information conveyed during the MCP initialization handshake (InitializeResult.serverInfo, InitializeResult.capabilities [6]) and the definitions of its offered tools, resources, and prompts.

MCP ToolDefinition, ResourceDefinition, and PromptDefinition can be quite detailed, often involving JSON Schemas or Zod schemas (in TypeScript implementations [30]) to describe parameters and structures. Storing these complete schemas on-chain for every tool, resource, and prompt offered by potentially many servers would be highly inefficient and costly due to Solana's storage limitations [7] and rent economics.[11]

Therefore, the McpServerRegistryEntryV1 adopts a hybrid on-chain/off-chain data model:

1. **Core Server Information On-Chain:** Key identifying details such as server ID,

name, primary service endpoint, version, status, and flags indicating support for tools, resources, and prompts (derived from MCP ServerCapabilities [6]) are stored directly on-chain.
2. **Summaries and Hashes of Key Offerings:** For a limited number of flagship tools, resources, and prompts, the registry can store their names and cryptographic hashes (e.g., SHA256) of their detailed descriptions and input/output schemas. This allows for some level of on-chain verification and basic discovery of core functionalities.
3. **full_capabilities_uri for Comprehensive Details:** A crucial field, full_capabilities_uri, will point to an off-chain location (e.g., an IPFS-hosted JSON file). This JSON file will contain the complete, MCP-compliant definitions of *all* tools, resources, and prompts offered by the server, structured according to the MCP schema.ts [29] or its JSON Schema equivalent.[32] AI clients would fetch and parse this URI to obtain the full information needed to interact with the server's capabilities.

This approach ensures that the on-chain registry remains lightweight and verifiable, while still enabling access to the rich, detailed information required by MCP clients.

The following table details the proposed McpServerRegistryEntryV1 structure:

Table 2: MCP Server Registry Entry Data Structure (McpServerRegistryEntryV1)

Field Name	Solana (Rust) Type	Borsh Serialization Notes (Illustrative)	Description	MCP Schema Mapping	Constraints /Examples
bump	u8	Single byte	Bump seed for PDA derivation.	N/A	Required.
registry_version	u8	Single byte	Schema version (e.g., 1).	N/A	For upgradability.
owner_authority	Pubkey	32 bytes	Solana public key of the entry's owner.	N/A	Critical for access control.

server_id	String	Length-prefixed Vec<u8>	Unique identifier for the MCP server.	N/A (MCP server identity)	Max 64 chars.
name	String	Length-prefixed Vec<u8>	Human-readable server name.	InitializeResult.serverInfo.name [6]	Max 128 chars.
server_version	String	Length-prefixed Vec<u8>	Version of the MCP server software.	InitializeResult.serverInfo.version [6]	Max 32 chars.
service_endpoint	String	Length-prefixed Vec<u8>	Primary URL for MCP communication (HTTP/SSE).	MCP: JSON-RPC over HTTP(S) [3]	Max 256 chars, HTTPS REQUIRED.
documentation_url	Option<String>	Optional, then Length-prefixed Vec<u8>	URL to human-readable documentation.	N/A	Max 256 chars.
server_capabilities_summary	Option<String>	Optional, then Length-prefixed Vec<u8>	Brief summary of server offerings.	Derived from InitializeResult.instructions [6]	Max 256 chars.
supports_resources	bool	Single byte (0 or 1)	True if server offers MCP Resources.	ServerCapabilities.resources [6]	
supports_tools	bool	Single byte (0 or 1)	True if server offers MCP Tools.	ServerCapabilities.tools [6]	
supports_prompts	bool	Single byte (0 or 1)	True if server offers MCP Prompts.	ServerCapabilities.prompts [6]	

onchain_tool_definitions	Vec<McpToolDefinitionOnChain>	u32 count, then each struct	Summary of key on-chain advertised tools.	MCP Tool concept [27]	Max 5 tools on-chain.
↳ name	String	Length-prefixed Vec<u8>	Tool name.	ToolDefinition.name	Max 64 chars.
↳ description_hash	[u8; 32]	32 bytes	SHA256 hash of ToolDefinition.description.	ToolDefinition.description (hashed)	
↳ input_schema_hash	[u8; 32]	32 bytes	SHA256 hash of ToolDefinition.inputSchema.	ToolDefinition.inputSchema (hashed)	
↳ output_schema_hash	[u8; 32]	32 bytes	SHA256 hash of ToolDefinition.outputSchema.	ToolDefinition.outputSchema (hashed)	
↳ tags	Vec<String>	u32 count, then each length-prefixed Vec<u8>	Tags for the tool.	N/A	Max 3 tags, each max 32 chars.
onchain_resource_definitions	Vec<McpResourceDefinitionOnChain>	u32 count, then each struct	Summary of key on-chain advertised resources.	MCP Resource concept [27]	Max 5 resources on-chain.
↳ uri_pattern	String	Length-prefixed Vec<u8>	Resource URI or pattern.	ResourceDefinition.uri	Max 128 chars.
↳ description_hash	[u8; 32]	32 bytes	SHA256 hash of ResourceDef	ResourceDefinition.description	

			inition.description.	(hashed)	
↳ tags	Vec<String>	u32 count, then each length-prefixed Vec<u8>	Tags for the resource.	N/A	Max 3 tags, each max 32 chars.
onchain_prompt_definitions	Vec<McpPromptDefinitionOnChain>	u32 count, then each struct	Summary of key on-chain advertised prompts.	MCP Prompt concept [27]	Max 5 prompts on-chain.
↳ name	String	Length-prefixed Vec<u8>	Prompt name.	PromptDefinition.name	Max 64 chars.
↳ description_hash	[u8; 32]	32 bytes	SHA256 hash of PromptDefinition.description.	PromptDefinition.description (hashed)	
↳ tags	Vec<String>	u32 count, then each length-prefixed Vec<u8>	Tags for the prompt.	N/A	Max 3 tags, each max 32 chars.
status	u8	Single byte	Server status (0:Pending, 1:Active, 2:Inactive, 3:Deprecated).	N/A	
registration_timestamp	i64	8 bytes (Unix timestamp)	Timestamp of initial registration.	N/A	
last_update_timestamp	i64	8 bytes (Unix timestamp)	Timestamp of the last update.	N/A	
full_capabiliti	Option<Strin	Optional,	URI to	N/A	Max 256

es_uri	g>	then Length-prefixed Vec<u8>	off-chain JSON with full Tool/Resource/Prompt definitions (matching MCP schema.ts [29]).		chars (e.g., IPFS, Arweave).
tags	Vec<String>	u32 count, then each length-prefixed Vec<u8>	General discoverability tags for the server.	N/A	Max 10 tags, each max 32 chars.

Registration and Management Instructions (Solana Program Functions)

The MCP Server Registry program will provide instructions analogous to those in the Agent Registry for managing MCP server entries. These instructions will handle the creation, updating, and deletion of PDAs storing McpServerRegistryEntryV1 data. Crucially, all modification instructions must be signed by the owner_authority of the respective entry.

- **register_mcp_server(ctx, server_id: String, name: String,..., full_capabilities_uri: Option<String>)**: Creates and initializes a new MCP server entry PDA. Emits McpServerRegistered.
- **update_mcp_server_details(ctx, server_id: String, new_name: Option<String>,...)**: Modifies fields of an existing MCP server entry. Emits McpServerUpdated.
- **update_mcp_server_status(ctx, server_id: String, new_status: u8)**: Changes the server's operational status. Emits McpServerStatusChanged.
- **deregister_mcp_server(ctx, server_id: String)**: Marks an MCP server as deregistered or closes its PDA. Emits McpServerDeregistered.

Discoverability and Querying Mechanisms (MCP Server Registry)

The discovery mechanisms for MCP servers mirror those of the Agent Registry, emphasizing a hybrid on-chain/off-chain approach:

- **Direct Lookup**: Possible if server_id is known and used as a PDA seed.
- **On-Chain Filtering (Limited)**:
  - Filtering by boolean flags like supports_tools, supports_resources, or supports_prompts is feasible if these flags are part of the PDA seed structure

(though this complicates seed design) or by client-side filtering of getProgramAccounts results.
    - Discovering servers based on specific tool names or resource URI patterns on-chain is generally impractical due to the complexity and volume of this data.
- **Event Emission for Off-Chain Indexing**: This is paramount for meaningful discovery of MCP servers based on their specific offerings.
    - McpServerRegistered: Emits the full McpServerRegistryEntryV1 data, including the on-chain summaries and, critically, the full_capabilities_uri.
    - McpServerUpdated: Details the server_id and changed fields.
    - McpServerStatusChanged: server_id and new status.
    - McpServerDeregistered: server_id.

Off-chain indexers will subscribe to these events. Upon receiving an McpServerRegistered or McpServerUpdated event, the indexer will fetch the JSON data from the full_capabilities_uri. This JSON file will contain the detailed arrays of ToolDefinition, ResourceDefinition, and PromptDefinition objects, as specified by the MCP schema.ts.[29] The indexer can then parse these definitions (including tool names, resource URIs, prompt names, descriptions, input/output schemas, and associated tags) and store them in its own database. This enables users to perform rich queries such as "find all MCP servers that offer a tool named 'ImageAnalysisTool' which accepts 'image/png' as input" or "list MCP servers providing resources matching the URI pattern 'crm://contacts/*'". The on-chain registry provides the pointers (URIs) and essential summaries, while the off-chain indexer provides the deep queryability into the server's full capabilities.

5. Advanced Discoverability, Search, and Querying Mechanisms (Cross-Registry)

While individual registries provide foundational discovery, advanced use cases often require searching across registries or using complex criteria that go beyond simple on-chain lookups. This section explores strategies for enhancing discoverability for both the Agent and MCP Server registries.

Designing PDA Structures for Querying

The primary PDA for an entry (e.g., AgentRegistryEntryV1 or McpServerRegistryEntryV1) is typically derived from a unique identifier for that entry (e.g., agent_id or server_id) and a prefix indicating the registry type. For instance:

- Agent PDA seeds: [b"agent_reg_v1", agent_id.as_bytes()]

- MCP Server PDA seeds: [b"mcp_srv_reg_v1", server_id.as_bytes()]

This structure allows for direct, O(1) lookup if the entry_id is known. However, it does not inherently support querying by other attributes stored within the PDA's data field.

Implementing On-Chain Secondary Indexes

To enable some level of on-chain querying by attributes other than the primary ID, on-chain secondary indexes can be implemented. This pattern involves creating additional PDAs whose existence or content links an attribute value to one or more primary registry entries.

- **Pattern Example (Agent Tag Index)**: To find agents by a specific tag (e.g., "finance"):
  - For each agent registered with the tag "finance", an additional "index PDA" could be created with seeds like [b"idx_agent_tag", b"finance", agent_id.as_bytes()].
  - The data stored in this index PDA might be minimal or even empty; its existence signifies the association.
  - To query for all agents tagged "finance", a client would use getProgramAccounts with a seed prefix filter matching [b"idx_agent_tag", b"finance"]. This would return all such index PDAs. The client would then extract the agent_id from the seeds of each returned index PDA and perform a primary lookup for each agent's full details.
- **Challenges with On-Chain Secondary Indexes**:
  - **Management Complexity**: Index PDAs must be created, updated, and deleted atomically with their corresponding primary entries. This adds complexity and CU cost to the main registration instructions.
  - **Scalability**: For attributes with high cardinality (many unique values, e.g., very specific skill names) or for entries associated with many indexed attributes, the number of index PDAs can proliferate, increasing storage costs.
  - **Storing Lists**: If an index PDA attempts to store a list of primary keys (e.g., a Vec<Pubkey> for a tag that maps to multiple agents), it can quickly hit Solana's account size limits (10MB [7]) or become inefficient to manage.[17] While linked lists of PDAs can mitigate the single-account size limit, they further increase complexity.

On-chain secondary indexes are feasible for a few, well-defined, low-cardinality attributes. However, they do not scale well for arbitrary, complex queries and add significant on-chain overhead. Their utility is often limited to enabling slightly more

granular filtering via getProgramAccounts than fetching all entries.

Strategies for Complex Queries (Primarily Off-Chain)

For the vast majority of complex search requirements—such as multi-field filtering (e.g., "find active agents with skill X and tag Y, registered in the last month"), full-text search on descriptions, or sorted results—off-chain solutions are indispensable.

Off-chain indexers [16] are purpose-built for this. They operate by:

1. Subscribing to on-chain events emitted by the registry programs (e.g., AgentRegistered, McpServerToolAdded).
2. Deserializing the event data, which includes the core on-chain information for the entry.
3. Fetching any linked off-chain metadata from URIs provided in the event (e.g., the extended_metadata_uri for agents or full_capabilities_uri for MCP servers). This off-chain data contains the rich, detailed descriptions, skill lists, tool schemas, etc.
4. Storing this aggregated on-chain and off-chain data in an optimized database (e.g., SQL, NoSQL, graph databases like Neo4j, or search engines like Elasticsearch).
5. Exposing a rich query API (e.g., GraphQL, REST) over this database, allowing for complex searches, aggregations, and sorting that are not feasible on-chain.

Role of Off-Chain Indexers and Data Providers

Several existing services and technologies can facilitate off-chain indexing for Solana programs:

- **RPC Providers with Enhanced APIs**: Some RPC providers offer enhanced APIs beyond the standard Solana RPC methods, potentially including some level of data indexing or filtering. Helius, for example, provides parsed transaction data and webhooks that can simplify the process of capturing relevant on-chain changes.[16]
- **The Graph**: A decentralized indexing protocol that allows developers to define "subgraphs" to index specific on-chain data and events, exposing it via a GraphQL API.
- **Custom Indexers**: Developers can build custom indexers using Solana's Geyser plugin interface [16], which allows direct streaming of account and transaction updates from a validator. This data can then be fed into any custom database and processing pipeline.
- **Data Platforms**: Services like Dune Analytics or Flipside Crypto also index Solana

data and provide SQL interfaces for querying [16], though these are more geared towards analytics than real-time application-level querying.

The design of the on-chain registry program's events is the critical interface to these off-chain systems. Comprehensive, well-structured, and reliably emitted events are the lifeblood of effective off-chain indexing and discoverability.

Standardized Event Logging for Discoverability

To maximize the utility for off-chain indexers, event logging from the registry programs must be standardized and comprehensive.

- **Event Structure**: Define the exact JSON (or Borsh-serialized, then base64 encoded for logs) structure for each event type (e.g., AgentRegistered, McpServerRegistered, AgentSkillAdded, McpServerToolUpdated).
- **Self-Contained Events**: Events should ideally contain all necessary information for an indexer to process the update without necessarily requiring an immediate follow-up getAccountInfo RPC call for the core data. This includes all on-chain fields and, crucially, the URIs for any extended off-chain metadata.
- **Versioning**: Event schemas should be versioned to allow for future evolution without breaking existing indexers.
- **Emission Reliability**: For critical events that must not be missed by indexers (e.g., new registrations), using Anchor's emit_cpi! [21] is recommended over emit!.[22] While emit_cpi! consumes more Compute Units as it involves a Cross-Program Invocation to log the event data as part of instruction data, it is less prone to log truncation issues that can occur with RPC providers handling standard program logs. This ensures that indexers receive a complete and reliable stream of changes.

The table below compares on-chain and off-chain querying strategies:

Table 3: Comparison of On-chain vs. Off-chain Querying Strategies

Query Type/Complexity	On-Chain Feasibility & Method	Off-Chain Feasibility & Method	Relative Performance (On-Chain / Off-Chain)	Relative Complexity to Implement (On-Chain / Off-Chain)	Typical Use Cases
Direct ID Lookup	High (PDA derivation +	High (DB primary key	Fast / Very Fast	Low / Medium	Fetching a known

	getAccountInfo)	lookup)			agent/server.
Single Attribute Filter (e.g., by status, by a specific tag if indexed on-chain)	Medium (PDA seed prefix + getProgramAccounts, then client filter)	High (DB query with WHERE clause)	Slow-Medium / Very Fast	Medium-High / Medium	Finding all "active" agents; finding agents with a common, indexed tag.
Multi-Attribute Filter (e.g., status AND tag AND skill)	Low (Complex multi-seed PDAs or full client-side scan of all entries)	High (Complex SQL/NoSQL query)	Very Slow / Fast	Very High / Medium	Advanced discovery, e.g., "find active MCP servers with tool X and tag Y".
Full-Text Search (e.g., on descriptions)	Very Low (Impractical)	High (Search engine like Elasticsearch)	N/A / Fast	N/A / High	Searching for keywords in agent/server descriptions.
Sorted Results (e.g., by registration date, by name)	Very Low (Impractical)	High (DB query with ORDER BY)	N/A / Fast	N/A / Medium	Displaying ranked or ordered lists.
Geospatial Queries (if applicable)	Very Low (Impractical)	Medium-High (Spatial DB extensions)	N/A / Medium-Fast	N/A / High	Finding agents/servers in a geographic region (if location data is available).
Aggregations (e.g., count of agents	Low (Requires iterating and	High (DB aggregate functions)	Slow / Fast	Medium-High / Medium	Analytics, dashboard statistics.

per tag)	counting, or dedicated counter PDAs)				

This comparison highlights that while on-chain mechanisms provide the source of truth and basic lookup capabilities, the heavy lifting for advanced search and discovery must be delegated to off-chain systems that are optimized for such tasks.

6. Security Considerations for Registries

The security and trustworthiness of the Agent and MCP Server Registries are paramount for their adoption and utility. Security considerations span program-level vulnerabilities, data integrity, access control, and the broader implications of registering potentially untrusted entities.

Ownership and Access Control

A fundamental security requirement is that only the legitimate owner of a registry entry can modify or delete it.

- **owner_authority**: Each registry entry PDA (AgentRegistryEntryV1, McpServerRegistryEntryV1) stores an owner_authority (a Solana Pubkey).
- **Instruction-Level Enforcement**: All Solana program instructions that modify an entry (e.g., update_agent_details, deregister_mcp_server) MUST verify that one of the transaction's signers is the owner_authority associated with the target PDA. This is a standard Solana security pattern.
- **Key Management**: The security of an entry then depends on the owner_authority's private key management. Users must be educated on secure key practices. Multisig solutions could be used to manage owner_authority for high-value entries.

Data Integrity and Validation

Ensuring the integrity and validity of data stored in the registry is crucial.

- **Input Validation**: The Solana program instructions MUST perform rigorous input validation for all data fields. This includes checking string lengths against defined maximums, validating URI formats, ensuring enum values are within permitted ranges, and verifying that boolean flags are correctly set.
- **Hash Verification for Off-Chain Data**: For data stored off-chain but linked via URIs (e.g., extended_metadata_uri, full_capabilities_uri, or hashes of detailed

descriptions/schemas like description_hash, input_schema_hash), the on-chain entry stores a cryptographic hash (e.g., SHA256) of the canonical off-chain data. Clients fetching this off-chain data can recompute its hash and compare it against the on-chain hash to verify its integrity and ensure it hasn't been tampered with since registration. This provides a trust anchor for off-chain content.

Preventing Spam and Malicious Registrations

Public, permissionless registries are susceptible to spam or the registration of malicious entities.

- **Registration Fees**: Requiring a nominal registration fee (payable in SOL or a designated utility token) for creating a new entry can deter spam by making it economically unviable for bulk malicious registrations. This fee could be burned, contributed to a treasury for registry maintenance, or used to fund community moderation.
- **Curation Mechanisms (Optional)**:
  - **Centralized Curator**: A designated entity could approve new registrations. This improves quality control but introduces centralization.
  - **DAO-Based Curation**: A Decentralized Autonomous Organization (DAO) could vote on new registrations or flag problematic entries. This is more decentralized but adds governance overhead.
- **Reputation and Attestation Systems**: While the registry itself might not implement a full reputation system, it can include fields to link to external attestation services or reputation scores (e.g., a URI pointing to a Verifiable Credential or a community review page). This allows users to perform their own due diligence. The registry primarily verifies ownership of the *entry*, not necessarily the good intent of the agent/server itself.

Upgradability of Registry Program

The Solana programs for the registries will themselves be upgradeable by a designated upgrade authority.

- **Secure Upgrade Authority Management**: The private key for the program's upgrade authority must be securely managed, potentially by a multisig wallet or a DAO, to prevent unauthorized upgrades.
- **Data Migration**: If a program upgrade involves changes to on-chain data schemas (e.g., adding new fields to AgentRegistryEntryV1), a clear data migration strategy must be planned and communicated. This might involve deploying a new version of the program and providing a mechanism for users to migrate their

existing entries to the new format, or designing schemas to be backward-compatible where possible.

The security of these registries is not merely a technical concern but a foundational element for building trust in the broader AI agent ecosystem on Solana. While the on-chain mechanisms can enforce data ownership and provide some validation, community vigilance and integration with broader identity and reputation systems will be essential for mitigating risks associated with interacting with registered agents and servers.

7. Deployment and Operational Considerations

Beyond the core protocol design, successful implementation and adoption of the Agent and MCP Server Registries require careful consideration of their deployment, ongoing governance, and the ecosystem tooling needed to support them.

Initial Setup and Governance

- **Program Deployment**: The compiled Solana programs for the Agent Registry and MCP Server Registry will be deployed to the Solana blockchain. The initial deployment will establish the program IDs, which will be crucial for all interactions.
- **Upgrade Authority**: The entity or entities controlling the upgrade authority for these programs must be determined. For a community-focused infrastructure, this authority might eventually be transitioned to a DAO or a foundation responsible for the long-term maintenance and evolution of the registries.
- **Protocol Parameters**: Initial parameters, such as registration fees (if any), maximum string lengths, or limits on the number of on-chain skill/tool definitions, will need to be set. A governance mechanism should be established for updating these parameters if necessary. This could range from a simple admin key in early stages to a more formal on-chain or off-chain voting process as the ecosystem matures.

Gas Optimization and Rent Management

- **Efficient Program Instructions**: The Rust code for the Solana programs must be written with gas (Compute Unit) efficiency in mind. This involves optimizing data access patterns, minimizing unnecessary computations, and choosing efficient data structures for on-chain state. Solana transactions have compute unit limits, and exceeding them will cause transaction failure.[7]
- **Rent-Exemption Costs**: Registrants (those creating entries for their agents or MCP servers) must provide sufficient lamports to make their respective PDA

accounts rent-exempt.[11] Client applications and documentation should clearly communicate these costs. The size of the data stored directly impacts the rent-exemption threshold, reinforcing the need for the hybrid on-chain/off-chain data model.

Client-Side Considerations

For the registries to be widely used, developers will need tools and libraries to simplify interaction.

- **SDKs/Libraries**: Client-side SDKs (e.g., in TypeScript/JavaScript, Python, Rust) should be developed to abstract the complexities of:
  - Deriving PDAs for registry entries.
  - Serializing and deserializing registry data (Borsh).
  - Constructing and sending transactions for registration, updates, and queries.
  - Fetching and verifying off-chain metadata from URIs (e.g., extended_metadata_uri, full_capabilities_uri).
  - Parsing on-chain events for real-time updates.
- **Handling Hybrid Data**: Client applications need to be designed to work with the hybrid data model. When displaying agent or MCP server details, they will fetch the core on-chain data and then, if necessary, resolve the off-chain URIs to get the complete picture. Hash verification of off-chain content is a critical client-side responsibility.
- **Query Interfaces**: While advanced querying relies on off-chain indexers, client libraries can provide convenient wrappers for common query patterns, potentially interacting with known public indexer APIs or allowing configuration for custom indexers.

The operational success of these registries will depend on a combination of robust on-chain programs, well-managed governance, and a supportive ecosystem of developer tools and off-chain services.

8. Conclusion and Future Directions

Summary of Protocol Designs

This report has detailed the design for two critical pieces of infrastructure for a decentralized AI ecosystem on Solana: an Agent Registry and an MCP Server Registry. Both protocols leverage Solana's Program Derived Addresses for unique, program-controlled entries and Borsh for efficient on-chain data serialization. A key design principle is the hybrid storage model, where essential verifiable data resides on-chain, while more extensive, detailed metadata (like full A2A AgentCards or

complete MCP tool/resource/prompt schemas) is stored off-chain (e.g., on IPFS/Arweave) and linked via URIs, with on-chain hashes ensuring integrity.

The Agent Registry incorporates concepts from AEA and Google's A2A protocol, focusing on agent identity, capabilities, service endpoints, and economic intent. The MCP Server Registry aligns with the Model Context Protocol, enabling the advertisement of servers based on the tools, resources, and prompts they offer. For both, discoverability is achieved through a combination of direct PDA lookups, limited on-chain filtering, and, most importantly, robust event emission designed to power sophisticated off-chain indexing and query services.

Impact on Solana AI Ecosystem

The successful implementation and adoption of these registries can significantly benefit the Solana AI ecosystem by:

- **Enhancing Discoverability**: Providing a standardized, central place to find and learn about available AI agents and MCP-compliant data/tool servers.
- **Fostering Interoperability**: By promoting adherence to common metadata schemas (A2A, MCP), the registries can facilitate easier integration between different AI components and applications.
- **Building Trust**: On-chain verification of ownership and data integrity (via hashes of off-chain content) can increase confidence in the registered entities.
- **Stimulating Innovation**: A discoverable and interoperable ecosystem lowers the barrier to entry for developers building new AI-powered applications and services on Solana, encouraging the creation of more complex and collaborative agent systems.

Potential Extensions

The proposed registries provide a solid foundation, but several future directions could further enhance their capabilities and impact:

- **On-Chain Reputation and Attestation Systems**: Integrating or linking to systems where agents and servers can accumulate reputation scores or verifiable attestations (e.g., for security audits, quality of service, community endorsements) would further enhance trust.
- **More Sophisticated On-Chain Indexing**: As Solana's own capabilities for on-chain querying and data handling evolve (e.g., through improvements in RPC filtering or new on-chain data structures), some indexing logic currently delegated off-chain might become feasible on-chain.
- **Native Decentralized Storage Integration**: The protocols could offer more

direct integration with decentralized storage solutions like IPFS or Arweave for the extended_metadata_uri and full_capabilities_uri, perhaps by standardizing content addressing or providing on-chain mechanisms to update these links securely.

- **Cross-Chain Discovery Mechanisms**: As AI agents and services become more chain-agnostic, exploring protocols or bridges that allow discovery of Solana-registered entities from other blockchains, and vice-versa, will be important.
- **Standardized Off-Chain Indexer Interfaces**: Defining a standard API for off-chain indexers that consume data from these registries could promote a competitive and interoperable market of discovery services.
- **Automated Verification Services**: Services could emerge that automatically verify aspects of registered agents/servers, such as endpoint liveness, adherence to advertised protocols (A2A, MCP), or the integrity of their off-chain metadata, and then post attestations back to the registry or a linked reputation system.

By addressing the fundamental need for service discovery and verification, these Solana-based registries can play a pivotal role in accelerating the development and adoption of decentralized AI, paving the way for a more intelligent, autonomous, and interconnected digital future.

Works cited

1. Autonomous Economic Agent (AEA) Meaning in Crypto - Tangem, accessed on May 23, 2025, https://tangem.com/en/glossary/autonomous-economic-agent-aea/
2. Artificial Superintelligence Alliance price today, FET to USD live price, marketcap and chart | CoinMarketCap, accessed on May 23, 2025, https://coinmarketcap.com/currencies/artificial-superintelligence-alliance/
3. Specification - Agent2Agent Protocol (A2A) - Google, accessed on May 23, 2025, https://google.github.io/A2A/specification/
4. google/A2A: An open protocol enabling communication and interoperability between opaque agentic applications. - GitHub, accessed on May 23, 2025, https://github.com/google/A2A
5. A beginners Guide on Model Context Protocol (MCP) - OpenCV, accessed on May 23, 2025, https://opencv.org/blog/model-context-protocol/
6. Specification - Model Context Protocol, accessed on May 23, 2025, https://modelcontextprotocol.io/specification/2025-03-26
7. Deep Dive into Resource Limitations in Solana Development — CU Edition - 57Blocks, accessed on May 23, 2025, https://57blocks.io/blog/deep-dive-into-resource-limitations-in-solana-development-cu-edition

8. What are Solana PDAs? Explanation & Examples (2025) - Helius, accessed on May 23, 2025, https://www.helius.dev/blog/solana-pda
9. Program Derived Addresses (PDAs) - Solana, accessed on May 23, 2025, https://solana.com/developers/courses/native-onchain-development/program-derived-addresses
10. Solana Account Model, accessed on May 23, 2025, https://solana.com/docs/core/accounts
11. An Introduction to the Solana Account Model | QuickNode Guides, accessed on May 23, 2025, https://www.quicknode.com/guides/solana-development/getting-started/an-introduction-to-the-solana-account-model
12. How to Deserialize Account Data on Solana | QuickNode Guides, accessed on May 23, 2025, https://www.quicknode.com/guides/solana-development/accounts-and-data/how-to-deserialize-account-data-on-solana
13. Serialize Custom Instruction Data for Native Program Development - Solana, accessed on May 23, 2025, https://solana.com/developers/courses/native-onchain-development/serialize-instruction-data-frontend
14. Create a Basic Program, Part 2 - State Management - Solana, accessed on May 23, 2025, https://solana.com/developers/courses/native-onchain-development/program-state-management
15. solana-labs/token-list: The community maintained Solana ... - GitHub, accessed on May 23, 2025, https://github.com/solana-labs/token-list
16. Analyzing Solana On-chain Data: Tools & Dashboards - Helius, accessed on May 23, 2025, https://www.helius.dev/blog/solana-data-tools
17. Ultimate Solana Optimization Guide 2024: Boost Performance & Efficiency, accessed on May 23, 2025, https://www.rapidinnovation.io/post/solana-optimization-and-best-practices-guide
18. How to set space for PDA that store a Vec of structs in data - Solana Stack Exchange, accessed on May 23, 2025, https://solana.stackexchange.com/questions/5605/how-to-set-space-for-pda-that-store-a-vec-of-structs-in-data
19. List of 12 Indexing Tools on Solana (2025) - Alchemy, accessed on May 23, 2025, https://www.alchemy.com/dapps/list-of/indexing-tools-on-solana
20. hsyndeniz/solana-indexer - GitHub, accessed on May 23, 2025, https://github.com/hsyndeniz/solana-indexer
21. Emit Events, accessed on May 23, 2025, https://www.anchor-lang.com/docs/features/events
22. Solana logs, “events,” and transaction history - RareSkills, accessed on May 23, 2025, https://www.rareskills.io/post/solana-logs-transaction-history
23. Autonomous Economic Agents (AEAs) - GitHub, accessed on May 23, 2025, https://github.com/fetchai/agents-aea/blob/main/docs/aeas.md

24. What are best practices for Solana program development?, accessed on May 23, 2025, https://solana.com/docs/toolkit/best-practices
25. Introduction | Innovation Lab Resources - Fetch.ai Innovation Lab, accessed on May 23, 2025, https://innovationlab.fetch.ai/resources/docs/intro
26. Model Context Protocol (MCP): A comprehensive introduction for developers - Stytch, accessed on May 23, 2025, https://stytch.com/blog/model-context-protocol-introduction/
27. mark3labs/mcp-go: A Go implementation of the Model Context Protocol (MCP), enabling seamless integration between LLM applications and external data sources and tools. - GitHub, accessed on May 23, 2025, https://github.com/mark3labs/mcp-go
28. MCP Server - Model Context Protocol, accessed on May 23, 2025, https://modelcontextprotocol.io/sdk/java/mcp-server
29. specification/schema/2025-03-26/schema.ts at main - GitHub, accessed on May 23, 2025, https://github.com/modelcontextprotocol/specification/blob/main/schema/2025-03-26/schema.ts
30. modelcontextprotocol/typescript-sdk: The official Typescript ... - GitHub, accessed on May 23, 2025, https://github.com/modelcontextprotocol/typescript-sdk
31. stars/stars/modelcontextprotocol/typescript-sdk.md at master · DavidWells/stars - GitHub, accessed on May 23, 2025, https://github.com/DavidWells/stars/blob/master/stars/modelcontextprotocol/typescript-sdk.md
32. modelcontextprotocol/schema/2024-11-05/schema.json at main - GitHub, accessed on May 23, 2025, https://github.com/modelcontextprotocol/specification/blob/main/schema/2024-11-05/schema.json
33. How to Build a Solana Data Dashboard with Dune - Helius, accessed on May 23, 2025, https://www.helius.dev/blog/how-to-build-a-solana-data-dashboard-with-dune