

POLITECNICO DI TORINO

Master of Science in Cybersecurity



Securing Agent-to-Agent Communication: enhancing A2A protocol with Decentralized Identifiers and Verifiable Credentials

Supervisor:

Prof. Fulvio VALENZA

Candidate:

Cristian CARALLO

Academic Year 2025/2026

Abstract

The increasing usage of AI agents drives the shift from individual agent-based systems toward multi-agent deployments. In this scenario, an open standard has emerged: Agent-to-Agent Protocol. Abbreviated A2A, it provides a communication layer between AI agents, ensuring interoperability regardless of their internal architecture or implementation details. However, since this protocol is relatively new, it also introduces practical security challenges, especially when it comes to the Identity and Access Management part.

To address these challenges, this thesis proposes a security framework for multi-agent architectures, based on Decentralized Identifiers and Verifiable Credentials. Indeed, the main contribution of this thesis is an approach to enhance the authentication phase of the A2A protocol by embedding DIDs to enable a cryptographically verifiable identity and using VCs to represent the permissions held by agents.

Acknowledgements

Questa tesi rappresenta il frutto del mio percorso di studi Magistrale e di una collaborazione con Spike Reply. Per questo desidero ringraziare il mio relatore, il Prof. Fulvio Valenza, e l'Ing. Ivan Aimale per la fiducia riposta in me e per avermi consentito di vivere questa esperienza appieno.

Un sentito ringraziamento va anche ai miei colleghi Dario, Gaetano, Gianmario e Matteo. Mi avete saputo guidare con grande professionalità e competenza durante l'intero percorso, dandomi sempre i giusti consigli, trasmettendomi la passione e la forma mentis indispensabili per lavorare in questo settore. Non avrei potuto desiderare di meglio.

Vorrei ringraziare la mia famiglia, la cui lontananza è stata il carburante per migliorarmi, trovando ogni giorno la motivazione per continuare nonostante le difficoltà.

Infine, concludo ringraziando tutti i miei amici, che hanno contribuito a rallegrare questi anni e che sono uno dei motivi per cui questa città è diventata casa.

Table of Contents

List of Figures	VIII
List of Tables	IX
1 Introduction	1
1.1 Thesis Structure	3
2 Self-Sovereign Identity and DIDComm v2 Messaging	5
2.1 Self-Sovereign Stack	5
2.1.1 JOSE family	6
2.2 Decentralized Identifiers	7
2.2.1 DID methods	8
2.2.2 DID architecture	10
2.2.3 Resolution and dereferencing	11
2.2.4 Verification relationships	12
2.3 Verifiable Credentials and Verifiable Presentations	13
2.3.1 Ecosystems and roles	14
2.3.2 Structure and data fields	14
2.3.3 Verifiable Presentations	15
2.3.4 Selective-disclosure	16
2.4 DIDComm v2 Messaging	17
2.4.1 Communication model	17
2.4.2 Parties and roles	18
2.4.3 Message model	18
2.4.4 Cryptographic packaging and message delivery	20
2.4.5 Security and privacy considerations	21
3 Agent2Agent and Model Context Protocols	24
3.1 Agent2Agent Protocol	24
3.1.1 Discovery and Extensions	25
3.1.2 Task-centric interaction flow	26

3.2	Model Context Protocol	28
3.2.1	MCP architecture	28
3.2.2	MCP server features	29
3.3	A2A and MCP as a Composed Architecture	29
3.3.1	Horizontal vs vertical integration	29
3.3.2	Bridging A2A coordination with MCP tooling	30
3.3.3	A2A and MCP sample skeletons	31
3.3.4	Comparative summary	32
3.4	Transporting DIDComm Envelopes over A2A	32
3.4.1	Correlation strategy	34
3.5	Security Considerations in A2A–MCP Systems	34
3.5.1	Threat surface and trust boundaries	34
3.5.2	Authentication and authorization	35
3.5.3	Tool hardening	35
3.5.4	Sensitive data handling	36
3.5.5	Governance controls	37
4	Thesis Objectives	38
5	Methodology and Architectural Design	41
5.1	Microsoft Entra Verified ID: Overview	41
5.1.1	Organization onboarding	41
5.2	Microsoft Entra Verified ID: Environment Setup	42
5.2.1	Tenant configuration	43
5.2.2	Credential contract	44
5.2.3	Attestation method	46
5.3	Digital Wallet Walt.id	47
5.4	Webhook Backend for Callbacks	49
5.4.1	Design and implementation	49
5.4.2	Service functionalities	50
5.5	Architectural Concept	51
5.6	End-to-end Workflow	52
6	System Implementation	54
6.1	AI Agents	54
6.1.1	Integration with walt.id wallet	54
6.1.2	Integration with the Model Context Protocol	56
6.2	DIDComm v2: Setup and Messaging	60
6.2.1	ResolversConfig: DID and Secrets resolution	60
6.2.2	Message pack and unpack workflow	62
6.3	A2A Agent Card Design and Authorization Mechanisms	65

6.3.1	Discovery via Agent Card	65
6.3.2	Authenticated Extended Agent Card	66
6.4	Router-Based A2A Execution Model	67
6.4.1	Router Agent Executor	67
6.5	Issuer Backend Application	69
6.5.1	Access token acquisition	70
6.5.2	Issuance request creation	70
6.5.3	OID4VCI issuance automation	72
6.6	Verifier Backend Application	72
6.6.1	OIDC4VP presentation automation	74
6.6.2	Admin access token grant	74
7	Validation	76
7.1	Authentication and Authorization Flows	76
7.1.1	A2A-DIDAuth authentication	77
7.1.2	Verifiable Credential based authorization	80
7.2	Application demo flow: DIDComm-secured weather request	81
7.3	Theoretical validation	82
7.4	Practical attack-oriented validation	84
7.4.1	A2A-DIDAuth impersonation attack	85
7.4.2	DIDComm recipient-key substitution through malicious DID resolution	86
7.4.3	Replay attempt on <code>/getAccessToken</code> endpoint	87
7.4.4	Replay of a stolen access token in the application flow	89
7.4.5	MCP tool-boundary enforcement	90
8	Conclusions	92
A	Sequence Diagrams	95
	Bibliography	102

List of Figures

2.1	Conceptual layers of an SSI ecosystem. Source: [1]	6
2.2	Example of a DID. Source: [2]	8
2.3	Overview of DID architecture. Source: [2]	10
2.4	Roles of a VC-based system. Source: [13]	14
2.5	Components of a Verifiable Credentials (left) and components of a Verifiable Presentation (right). Source: [3]	15
2.6	JWT vs. SD-JWT: plaintext claims replaced by hashed commitments in the SD-JWT payload. Source: [14]	16
2.7	Overview of a DIDComm communication flow. Source: [16]	18
2.8	DIDComm message routing via a mediator. Source: [15]	21
3.1	Overview of A2A Protocol core features. Source: [19]	25
3.2	Task states. Source: [22]	26
3.3	A2A example flow. Source: [23]	27
3.4	MCP client-server structure and example of capabilities. Source: [25]	28
3.5	Example of integrated A2A–MCP architecture. Source: [20]	30
5.1	Visual representation of the AdminCredential	45
5.2	Schematic representation of walt.id wallet role. Source: [38]	48
5.3	High-level architecture of the thesis prototype	51
6.1	Issuer application overview. Source: [38]	69
7.1	MCP boundary test response	91
A.1	Verifiable Credential Issuance flow	95
A.2	Verifiable Credential Presentation flow	96
A.3	Authenticated Extended Card retrieval flow	97
A.4	A2A-DIDAuth flow – part 1	98
A.5	VC-based authorization flow	100
A.6	Application demo flow: weather request over DIDComm channel	101

List of Tables

3.1	High-level comparison of A2A and MCP	32
-----	--	----

Listings

2.1	Example of a DID Document	13
3.1	Example of message/send skeleton	31
3.2	Example of MCP tool calls	32
5.1	Verifier/Issuer DID document	43
5.2	Display definition for AdminCredential	44
5.3	Rules definition for AdminCredential	45
6.1	WaltIdClient API call wrapper	55
6.2	MCP Identity Server spawn	56
6.3	MCP Identity server skeleton with FastMCP	57
6.4	Bob connection to the MCP Hub	57
6.5	MCP Hub skeleton	58
6.6	Bob LLM configuration via environment variables	59
6.7	Configuration of ResolversConfig for Agents	60
6.8	Static DID resolver	61
6.9	Secrets Resolver integrated with walt.id wallet	61
6.10	Crafting of a JWE DIDcomm message with pack_encrypted	62
6.11	Embedding DIDComm inside A2A	63
6.12	Bob operations on receiver side	64
6.13	Wrapping the DIDComm response as an A2A Task artifact	64
6.14	Bob Agent Card construction	65
6.15	Middleware enforcing Bearer token for agent/getAuthenticatedExtendedCard	66
6.16	Flow-discrimination functions	68
6.17	Delegation to the specialized executors	68
6.18	OAuth2 client_credentials against the v2 token endpoint	70
6.19	Creation of an issuance request	71
6.20	Issuance offer consumption via walt.id endpoint	72
6.21	Creation of a presentation request	73
7.1	Simplified A2A Server Agent Card	78
7.2	Snippet of the DIDAuth impersonation test	86
7.3	Recipient-key poisoning test for DIDComm packing/unpacking	87

7.4	Critical validation checks in <code>/getAccessToken</code> endpoint	88
7.5	Identity binding at token enforcement time	89
7.6	Weather-only MCP allowlist enforced by Bob	90

Chapter 1

Introduction

Large Language Model (LLM) based systems are increasingly being used as **AI agents**, i.e., software components that do more than just text generation, since they can understand goals, execute multi-step procedures, and interact with the outside world to achieve some result. In the enterprise domain, this can take many different forms, ranging from support agents and knowledge assistants for employees to business procedures that query data sources, execute actions, and generate reports. In such cases, the behavior of an agent is clearly not just a matter of dialogue; it is a matter of switching between **reasoning** and **acting** according to the "ReAct" paradigm, where "acting" typically means API calls, document retrieval, or invoking specialized services.

As applications become more complex, a monolithic single agent can become a performance bottleneck. More realistic applications can be achieved with multiple agents, with different agents optimized for different sub-tasks or domains/scopes, with a coordinator for task decomposition and aggregation. This aspect raises practical interoperability problem: *how can agents and tools developed independently be composed without tight coupling?* Two new standards have been developed to address this split directly. **A2A (Agent-to-Agent Protocol)** offers a task-centric layer for agent-to-agent delegation and status reporting, allowing heterogeneous agents to cooperate through a shared interaction contract. **MCP (Model Context Protocol)** offers a uniform interface for connecting AI applications to tools and resources such as databases, file repositories, search systems, and structured prompts with a clear client/server boundary. The combined effect is a more modular architecture for agent coordination that is independent of tool connectivity.

At the same time, modularity in itself does not guarantee safety. This is especially true when there is a chain of delegations involving multiple agents and tools. This causes the trust boundary to extend to an end-to-end pipeline. What may have been a tolerable threat in a single-service system can become more critical with a chain of delegations and tool calls. For instance, spoofing identities can

misroute actions, messages can be replayed to re-trigger privileged actions, and credentials can be leaked through the output of tools, logs, or even through the propagation of prompts. In a nutshell, while part of the security problem is *what an agent decides*, it is also *who is allowed to request an action, how this is verified, and how sensitive information is protected while the request is processed*.

There is a parallel evolution happening in **digital identity**. Traditional identity systems are often designed around **centralized infrastructure** such as Identity Providers or Certificate Authorities, which manage the binding of identifiers to keys and the issuance of assertions. This approach is operationally well understood but creates a central trust model and often leads to fragmented attribute management across domains. **Decentralized Identity** introduces a new trust model in which identifiers and assertions are verifiable using cryptography, not a central platform operator. In this new model, **Decentralized Identifiers (DIDs)** are resolvable identifiers bound to public key material, and **Verifiable Credentials (VCs)** are signed assertions usable as evidence of permissions or attributes.

A major obstacle, however, is that a number of enterprise-level credential flows are based on a **human-centric flows** model. This is where a user is required to install a digital wallet, and frequently, issuance and/or presentation is started by a QR code and is often completed by a smartphone-mediated interaction. This is perfectly fine for human users but is not suitable for an automated multi-agent system, where a credential should be able to become part of the agent’s runtime environment. In multi-agent automation, an agent should be able to request, store, and present credentials programmatically, without requiring a manual scan-and-confirm loop. This thesis has attempted to solve this problem by modifying a credential-based identity model, originally designed for human users, and making it suitable for automated systems and AI agent, while still maintaining the essential trust properties.

This thesis work presents an end-to-end approach to connect agent interoperability with decentralized identity guarantees. It starts with a security risk analysis of the combination of A2A delegation and MCP tool access, focusing on attacks and failure scenarios that are feasible in a distributed tool-assisted execution environment. It then presents a compositional approach to agent systems, which keeps communication between agents task-focused and implementation-independent, while keeping tool interactions and secrets local to the agent responsible for them. This limits the impact of tool compromise and avoids exposing tool internal state to the orchestrating layer.

To validate feasibility, the thesis implements a reference **prototype** in which authentication and authorization are brought into the agent workflow using decentralized identity primitives. A dedicated Python library, called **a2a-didauth**, enables mutual authentication based on DIDs by transporting the relevant identity material within A2A payloads, rather than relying exclusively on transport-level

identity assumptions. Authorization is expressed through verifiable credentials that can be presented and checked as part of task execution. After access is granted, a protected DIDComm v2 messaging channel can be established for subsequent exchanges, ensuring that sensitive task artifacts remain confidential and tamper-evident during execution.

On the enterprise credential side, a concrete issuer/verifier setup is realized using **Microsoft Entra Verified ID**. Public issuer artifacts are hosted via an Azure Storage endpoint, and an auxiliary verification backend is deployed via an Azure Application and exposed to agents as an MCP tool, so that credential-related checks become a controlled capability invoked by specialists without coupling the coordinator to backend verification logic.

Finally, the thesis proposes the **validation** phase for the prototype, both conceptually and experimentally. Conceptually, the secured design is compared to the baseline that does not have identity-grounded protections, emphasizing how the proposed mechanisms counteract the main risk categories. Experimentally, failure-oriented tests are carried out (for example, malformed identity content, invalid signatures, replayed challenges, and tampered tokens) to verify that the system correctly rejects dangerous inputs deterministically and indicates errors traceably. Overall, the thesis shows that A2A-based orchestration and MCP-based tool integration can remain interoperable and modular, while obtaining stronger guarantees when authentication, authorization, and protected messaging are grounded in decentralized identity constructs rather than in a single centralized trust anchor.

1.1 Thesis Structure

This thesis is organized to take the reader step by step, from the main idea and motivation to system design, implementation, and final testing. Indeed, the remainder of the thesis is structured in the following way.

Chapter 2: Self-Sovereign Identity and DIDComm v2 Messaging

This chapter introduces Self-Sovereign Identity as a model where people and organizations control their own identifiers and credentials, without relying on a central authority. It explains the main building blocks of SSI such as Verifiable Data Registries, Decentralized Identifiers, and Verifiable Credentials/Presentations. Finally, it presents DIDComm v2 as a secure, transport-independent messaging framework for peer-to-peer communication, including its message structure, encryption modes and key security and privacy considerations.

Chapter 3: Agent2Agent and Model Context Protocols

This chapter presents two open standards that help connect agentic systems: A2A for agent-to-agent collaboration and MCP for connecting agents to tools, data, and prompts. It explains the main concepts of each protocol and shows how they can work together in one architecture. Finally, it describes how DIDComm encrypted messages can be carried inside A2A payload, and it discusses key security points such as trust boundaries, authorization, tool isolation, and governance.

Chapter 4: Thesis objective

The chapter motivates the thesis by highlighting open issues in nowadays DID/VC and A2A/MCP systems. Then it argues that autonomous GenAI agents need portable, verifiable identity and authorization across heterogeneous environments, and finally it defines five concrete objectives the thesis achieves.

Chapter 5: Methodology and Architectural Design

In this chapter, the methodological approach and the general architecture of the proof of concept are described. The entities and services will be all explained, by also giving a glimpse of their main interactions. Finally, the entire process of the project will be shown.

Chapter 6: System Implementation

In this chapter, the realization of the proposed architecture within the prototype will be described. In particular, the main software components and their related tasks will be discussed. In particular, this chapter focuses on the agent-side building blocks, the A2A server surface, the router/executor model, and the implementation of the Issuer and Verifier application.

Chapter 7: Validation

This chapter validates the proposed prototype from both a theoretical and practical security perspective. It presents the main flows, focusing on DID-based mutual authentication and VC-based authorization, analyzes the security properties of the design, and reports practical attack and failure tests implemented with `pytest`.

Chapter 8: Conclusions

This final chapter summarizes the main outcomes of the thesis, highlights the key contributions of the proposed approach, and outlines the main limitations and possible directions for future work.

Chapter 2

Self-Sovereign Identity and DIDComm v2 Messaging

The aim of this chapter is to present theoretical aspects related to the concept of Self-Sovereign Identity, abbreviated as SSI, which, in essence, is a paradigm that offers the possibility of developing a different model of Identity and Access Management (IAM). In addition, the current chapter will also present DIDComm v2, which is a secure messaging protocol that relies on the principles of SSI, such as the utilization of DID and associated key material, in order to facilitate secure peer-to-peer communication.

2.1 Self-Sovereign Stack

Self-Sovereign Identity is an identity model in which trust is anchored in **cryptographic proofs** and **decentralized registries** rather than typical centralized identity providers [1]. In this model, entities have the power to decide what to disclose, use identity data across services and peer-to-peer interactions. SSI architecture is typically conceptualized as a robust, layered technology stack [1], with three interconnected layers.

At the foundational layer, there is the **Verifiable Data Registry (VDR)**, which is often built on top of blockchain or distributed ledger technology (DLT) and plays a vital function as the decentralized root of trust. Moreover, it provides a highly available, tamper-evident lookup for public identifiers, schema definitions, and revocation registries with the goal of eliminating the need for centralized Certificate Authorities (CAs) and administrative hurdles.

Based on this foundation, there is the layer of **Decentralized Identifiers (DIDs)** [2]. DIDs are cryptographically verifiable, globally unique, and resolvable identifiers that are entirely independent of any centralized registry, identity provider,

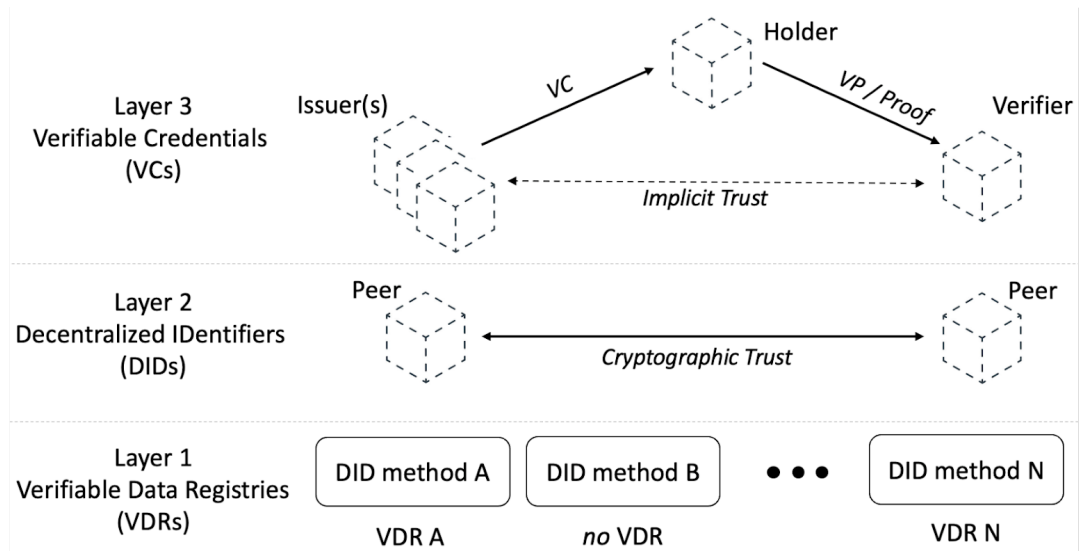


Figure 2.1: Conceptual layers of an SSI ecosystem. Source: [1]

or certificate authority. A DID resolves to a DID Document, a data structure containing the public cryptographic material necessary to interact with the identity subject. This architecture ensures that the DID controller retains full sovereignty over their identifier without risk of administrative lock-out.

The application layer is populated by **Verifiable Credentials (VCs)** [3]. These can be considered the digital equivalent of physical credentials such as passports or university degrees, with the added advantage of being cryptographically signed by an issuer. VCs enable the Trust Triangle interaction model between the Issuer, Holder, and Verifier. However, the distinction between the key material and control, and the identity attributes, creates significant architectural advantages. For instance, it allows for key rotation without the need to re-issue identity claims and supports privacy-preserving disclosure patterns. For example, through the use of mechanisms such as selective disclosure and ZKPs, the holder can disclose attributes to the verifier without revealing the entire contents of the credential.

2.1.1 JOSE family

In the SSI stack, information is transferred in JSON documents such as DID Documents, Verifiable Credentials, and messages sent over various protocols. It's therefore crucial that widely accepted and standardized security mechanisms are employed in order to securely transport JSON documents over untrusted networks.

The **JavaScript Object Signing and Encryption (JOSE)** family provides exactly this: «a practical toolbox using JSON-based representations to represent

keys, sign data, and encrypt content in a way that different systems can implement consistently» [4]. JOSE, at its heart, is based on JSON as the data model for both claims and metadata and introduces standardized ways of protecting integrity and confidentiality, as well as interoperable key representations.

In practice, JOSE components are commonly combined as follows:

- **JSON Web Key (JWK)**: is the JSON data structure used to represent cryptographic keys (and JWK Sets to represent collections of keys). JWK enables interoperable distribution and consumption of key material for JOSE operations (e.g., verifying signatures or decrypting ciphertext) [5].
- **JSON Web Signature (JWS)**: provides integrity protection for content using either a digital signature or a MAC. JWS defines a compact, URL-safe representation (widely used in HTTP headers) and a JSON serialization that supports multiple signatures. Algorithm identifiers are defined separately, while JWS standardizes the signed structure and processing rules [6].
- **JSON Web Encryption (JWE)**: provides confidentiality by encrypting the content (e.g., a set of claims), producing an opaque token for parties that do not possess the appropriate decryption key [7].
- **JSON Web Token (JWT)**: is a compact, URL-safe container for claims represented as a JSON object. A JWT is realized by embedding the claim set as the payload of a JWS (signed/MACed) or as the plaintext of a JWE (encrypted), thereby connecting the token format with JOSE protection mechanisms [8].

Overall, JOSE standardizes how claims are modeled as JSON objects, how they are protected via signing (JWS) and/or encryption (JWE), and how the necessary key material is expressed and exchanged (JWK), with JWT serving as the most common “packaging” format for these protected claims.

This is particularly important in the context of DIDComm because DIDComm messages are expressed as JSON, but their security envelope is typically provided using JOSE structures: encrypted DIDComm messages are carried as JWE (`application/didcomm-encrypted+json`) and, when required, signed messages are carried as JWS, enabling interoperable end-to-end confidentiality and integrity/authenticity even across untrusted intermediaries.

2.2 Decentralized Identifiers

A Decentralized Identifier (DID) is a **globally unique URI** that can be created and controlled without a centralized registration authority [9]. It’s structured as

`did:method:specific-id` and it identifies a subject such as a person, organization, device, service, or abstract entity.

It can be resolved to a **DID Document** containing the cryptographic material and associated metadata needed to authenticate the DID controller and interact with the subject via advertised service endpoints. In this model, control is demonstrated cryptographically rather than delegated to a third-party identity provider.

The base DID can be extended into a **DID URL** by adding standard URI components (path, query, fragment) to reference a more specific target, such as a particular verification key within the DID Document. Instead, the **DID method** specifies the concrete rules that govern the DID management process and defines the corresponding registry and validation procedures that must be applied.

2.2.1 DID methods

A Decentralized Identifier (DID) is a URI with the generic structure shown in Fig. 2.2:



Figure 2.2: Example of a DID. Source: [2]

The **method** selects a *DID method specification*, which defines the concrete rules for the full DID lifecycle: creation, resolution (read), update (e.g., key rotation), and deactivation. In practice, this means that the method determines:

- where the information needed to construct the DID Document is stored (e.g., a web location, a ledger, or self-contained data),
- which algorithm a resolver must apply to obtain a DID Document that conforms to the method

Example: `did:web`

The `did:web` method binds a DID to control over a web domain (and optionally a path) by publishing the DID Document at a well-known HTTPS location [10]. The method-specific identifier is a fully qualified domain name, optionally followed by path segments encoded with colons (“:”), plus optional port handling via percent-encoding.

Concretely, resolution works by translating the DID into an HTTPS URL and downloading a `did.json` file:

1. Start from a DID such as `did:web:example.com` or `did:web:example.com:user:alice`
2. Convert the method-specific identifier by replacing each “:” (used as a path separator) with “/”
3. Build the HTTPS URL:
 - If the DID contains only a domain, fetch `/.well-known/did.json`
 - If the DID includes a path, fetch `/.../did.json` under that path
4. Perform an HTTPS GET, and verify that the retrieved DID Document has an `id` matching the DID being resolved

For instance, basing on the previous examples:

- `did:web:example.com` → `https://example.com/.well-known/did.json`
- `did:web:example.com:user:alice` → `https://example.com/user/alice/did.json`

Example: `did:jwk`

The `did:jwk` method is designed for scenarios where the public key material can be carried directly in the identifier, so that a DID Document can be derived without consulting an external registry, contrary to what happens in `did:web`. In this method, the method-specific identifier is obtained by encoding a JSON Web Key (JWK) [11].

A common creation procedure, as described in the official specification [12], is the following:

1. generate (or load) a JWK;
2. serialize the JWK to a UTF-8 string;
3. encode the resulting bytes using `base64url`;
4. prepend the `did:jwk:` prefix.

During resolution, a resolver decodes the `base64url`-encoded JWK from the DID and deterministically constructs a DID Document that exposes that key as a verification method [12]. Because the identifier effectively commits to a specific key, `did:jwk` is typically presented as a good fit when offline resolution is desirable and when key rotation and service endpoints are not required.

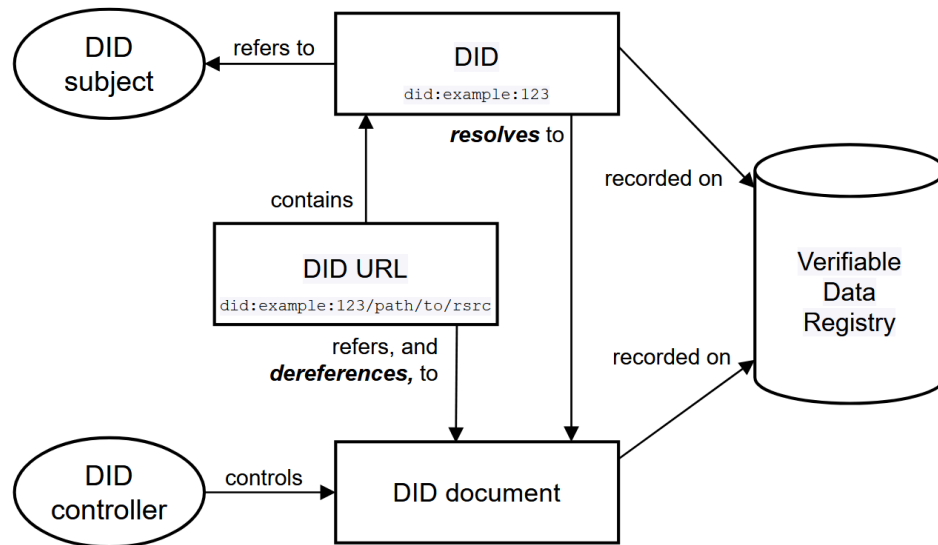


Figure 2.3: Overview of DID architecture. Source: [2]

2.2.2 DID architecture

This subsection offers a high-level view of the main building blocks in the Decentralized Identifier (DID) architecture, following the terminology of the W3C DID Core specification [2]. In essence, DIDs and DID Documents are anchored (in a method-dependent way) to a *Verifiable Data Registry*. A DID can be *resolved* to obtain its DID Document; a DID identifies a *subject*; and one or more *controllers* are able to update the DID Document according to the rules of the chosen DID method. In addition, a DID can be extended into a *DID URL* to point to a specific fragment within the DID Document or to a related external resource.

- **DIDs and DID URLs.** A Decentralized Identifier (DID) is a URI with three conceptual parts: the `did:` scheme, a method name, and a method-specific identifier (e.g., `did:method:identifier`). Through DID resolution, a DID can be turned into a corresponding DID Document. A DID URL builds on a DID by adding standard URI components (such as path, query, and fragment). This enables addressing a more specific target, for instance a particular verification key referenced inside the DID Document.
- **DID subjects.** The subject is whatever the DID is intended to denote. Depending on the use case, the subject can be a person, an organization, a device, a service, a group, or even a more abstract entity. The subject and the controller can coincide, but they do not have to.
- **DID controllers.** A controller is the party (human, organization, or software

agent) that is able to perform authorized changes to the DID Document, as permitted by the DID method. In most deployments this authority is enforced by control over cryptographic keys, although methods may support other control mechanisms. A DID may be managed by multiple controllers, and the subject may be one of them.

- **Verifiable Data Registries (VDRs).** For a DID to be resolvable, some underlying system must store (or make retrievable) the information required to construct the DID Document. Any system offering this function is called a Verifiable Data Registry. The term is **technology-agnostic** and can include distributed ledgers, decentralized storage, conventional databases, peer-to-peer networks, or other trusted storage substrates. A key property of this element, essential in SSI architectures, is that the registry used as a trust anchor is verifiable: anyone can independently validate that a DID Document is the correct one for a given DID, according to the rules of the underlying DID method and registry.
- **DID Documents.** A DID Document is a structured representation that associates metadata with a DID. Typically, it includes verification material (e.g., public keys) and service descriptors that support interaction with the subject (or its agent). DID Documents can be represented in multiple serializations and can change over time (e.g., key rotation), subject to the operations and constraints defined by the DID method.

2.2.3 Resolution and dereferencing

In DID-based systems, two related operations are central to retrieving information from an identifier: **DID resolution** and **DID URL dereferencing**. Both start from DID syntax, but they differ in their inputs and in the specificity of their outputs.

DID resolvers and DID resolution. A DID resolver is a software component (e.g., a library or a network service) that takes a bare DID as input and produces the corresponding DID Document as output, ensuring the result conforms to the rules of the relevant DID method. This transformation from DID to DID Document is called DID resolution. The applicable method specification determines the concrete resolution procedure and may also define method-specific metadata returned alongside the document.

DID resolution vs. DID URL dereferencing. While resolution answers the question “*What is the DID Document for this identifier?*”, DID URL dereferencing targets a more precise resource. A DID URL is a DID extended with standard URI

components (such as a fragment, path, and/or query), and a DID URL dereferencer is the component that accepts such a DID URL and returns the specific target it denotes. The dereferenced result can be an internal fragment of the DID Document such as a particular verification method, or an external resource referenced from the DID Document (e.g., a service endpoint or linked object). In other words, dereferencing answers: *“Given this DID URL, what exact resource does it point to?”*

Concrete example. Assume that resolving `did:example:123` yields a DID Document that includes a verification method identified as `did:example:123#key-1`. Then:

- Performing **DID resolution** on `did:example:123` returns the entire DID Document, including all verification methods, services, and relationships.
- Dereferencing the DID URL `did:example:123#key-1` returns only the specific key description addressed by the fragment `#key-1`, rather than the full document.

Conceptually, dereferencing can be seen as a two-step process: first resolve the DID to obtain the DID Document, then use the additional URL components (fragment/path/query) to select or retrieve the precise referenced resource.

2.2.4 Verification relationships

As already stated, the DID Document is a machine-readable resource describing how to verify proofs produced by the DID controller and how to interact with services associated with the DID. Although concrete fields differ across methods and profiles, DID Documents commonly contain:

- **Verification methods:** references to public keys (or other verification material), each with a unique identifier within the DID Document.
- **Verification relationships:** lists that indicate which verification methods are valid for specific purposes. Common examples include:
 - **authentication:** methods that can authenticate the controller (e.g., signing challenges during an authentication protocol);
 - **assertionMethod:** methods that can create assertions (e.g., signing credentials or presentations);
 - **keyAgreement:** methods used for key agreement / secure channel establishment. This property is particularly useful when encrypting messages for the DID subject. In this scenario, the counterparty uses the public

key found in the verification method to wrap a decryption key for the recipient;

- **Service endpoints:** metadata enabling interaction with the DID subject (e.g., an HTTPS endpoint, a messaging service, or an agent endpoint).

The snippet in Lst. 2.1 illustrates the shape of a DID Document.

```
1 {
2   "id": "did:example:123",
3   "verificationMethod": [
4     {
5       "id": "did:example:123#key-1",
6       "type": "...",
7       "controller": "did:example:123",
8       "publicKey...": "..."
9     }
10  ],
11  "authentication": ["did:example:123#key-1"],
12  "keyAgreement": ["did:example:123#key-1"],
13  "assertionMethod": ["did:example:123#key-1"],
14  "service": [
15    {
16      "id": "did:example:123#svc-1",
17      "type": "...",
18      "serviceEndpoint": "https://..."
19    }
20  ]
21 }
```

Listing 2.1: Example of a DID Document

2.3 Verifiable Credentials and Verifiable Presentations

DID Documents are specifically focused on control and interaction metadata, such as verification methods and service endpoints, and therefore do not directly “bind” the DID to real-world attributes such as name, student status, or organizational membership. This is where the need for Verifiable Credentials [3] is addressed, as it provides the ability for the issuer to cryptographically assert claims about the subject, where the subject is often represented by the DID, without the need of having a continuously online Identity Provider.

2.3.1 Ecosystems and roles

A Verifiable Credential is «a standardized, tamper-evident container for claims about a subject, protected by cryptographic proofs» [9]. Systems based on Verifiable Credentials typically involve three different and distinct roles:

- **Issuer:** creates a credential and applies a proof mechanism (e.g., a digital signature) attesting the claims about one or more subjects;
- **Holder:** an entity that stores the credential (e.g., in a digital wallet) and decides when and how to present it;
- **Verifier:** receives a credential (or a presentation derived from it) and validates integrity, authenticity, and additional acceptance conditions.

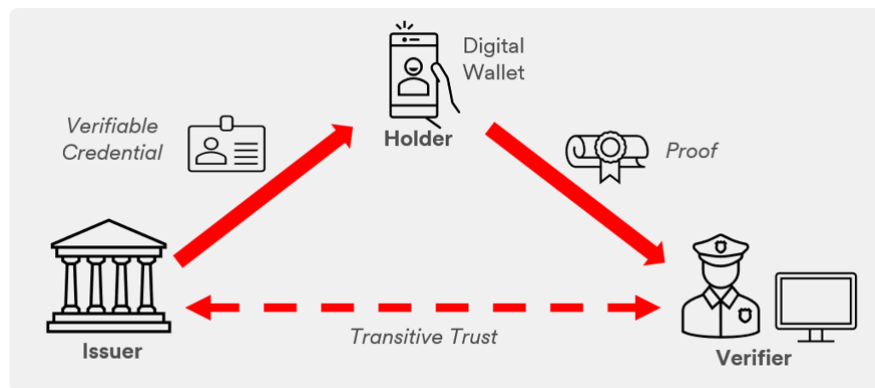


Figure 2.4: Roles of a VC-based system. Source: [13]

A key property is **asynchronous verification**: after issuance, a holder can present a VC to any verifier, who can validate it using the issuer's verification material that is commonly retrieved by resolving the issuer's DID.

Outside of identification, VCs can also support **authorization by attribute**. For example, instead of disclosing a full identity, a holder may demonstrate satisfaction of a condition such as "has role" (*employee*) or "has right" (*admin*).

2.3.2 Structure and data fields

VCs can be expressed in multiple representations (e.g., JSON-LD-based or JWT-based). Independent of the concrete encoding, widely used VC profiles share a common set of conceptual fields:

- **Context:** a list of references defining the vocabulary used in the credential;

- **Identifier and type:** an optional unique identifier and one or more types describing the credential class;
- **Issuer:** the identifier of the issuing entity (often a DID, sometimes a URL), used to retrieve the verification key material;
- **Credential subject:** the entity (or entities) the claims refer to, commonly identified by a DID plus one or more asserted attributes;
- **Validity interval:** temporal constraints such as an issuance time and an expiration time;
- **Schema and status (optional):** pointers to a schema/validation mechanism and revocation or suspension information (e.g., via a status list);
- **Proof:** cryptographic protection ensuring integrity and issuer authenticity.

A VC may refer to multiple subjects (e.g., a credential about more than one entity), and even a single claim can naturally involve multiple subjects (e.g., a credential analogous to a marriage certificate).

2.3.3 Verifiable Presentations

A Verifiable Presentation (VP) is a package created by the holder for a particular interaction. It may contain one or more VCs, or selectively derived statements from the credentials, and may also include a proof created by the holder to prove control or possession, whether necessary for the protocol.

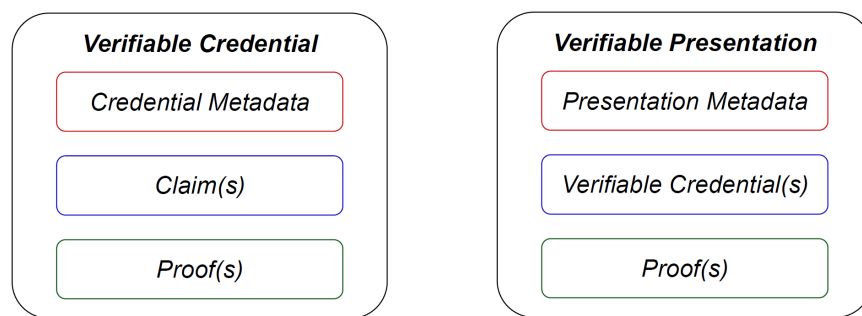


Figure 2.5: Components of a Verifiable Credentials (left) and components of a Verifiable Presentation (right). Source: [3]

2.3.4 Selective-disclosure

The primary motivation for VPs is **minimum disclosure**: in many real-world interactions, the verifier needs only a small subset of attributes (e.g., “is over 18”), while the original credential may contain additional identifying information. A VP enables the holder to provide only what is necessary, while preserving verifiability of what is disclosed.

One way to achieve the principle of minimum disclosure is through **Selective Disclosure**. This concept can be implemented using various cryptographic methods. A practical and commonly adopted approach is the **Selective Disclosure JWT (SD-JWT)**, which enables a holder to disclose only specific attributes while maintaining cryptographic verifiability.

Conceptually, an SD-JWT-based credential can be seen as the combination of the following base components:

- a **signed JWT payload** where claim values are not stored in cleartext but replaced by **hash commitments** (one per claim);
- a **signature** over the JWT (issuer authenticity and integrity);
- a **set of disclosures**, i.e., the plaintext claim values the holder decides to reveal, each packaged together with the random salt needed to validate the corresponding commitment.

The intuition is illustrated in Fig. 2.6 with a representative use case.

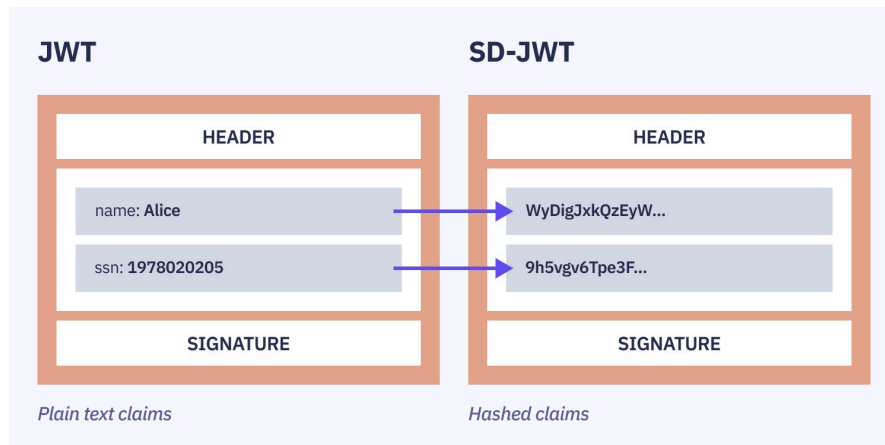


Figure 2.6: JWT vs. SD-JWT: plaintext claims replaced by hashed commitments in the SD-JWT payload. Source: [14]

In a traditional JWT, the payload contains plaintext claims (e.g., `name: Alice` and `ssn: 1978020205`). In an SD-JWT, those plaintext values are replaced by

hashed representations: for each claim, the issuer samples a fresh random salt and computes a hash over (salt, claim name, claim value). Only these hashes are embedded in the signed JWT payload.

During presentation, the holder transmits the signed SD-JWT, which contains hash commitments for all claims (e.g., both **name** and **ssn**), together with disclosures only for the attributes that should be revealed.

For example, if the verifier requires the subject's name but not the social security number, the holder supplies only the **name** disclosure (i.e., the plaintext value together with its associated salt) and omits the **ssn** disclosure. The verifier then first validates the issuer's digital signature on the SD-JWT and subsequently recomputes the corresponding hash commitments using the received salts and attribute values. By comparing these recomputed hashes with the hashed entries in the SD-JWT payload, the verifier confirms that the disclosed **name** is authentically attested by the issuer, while gaining no information about the undisclosed **ssn**.

Moreover, the use of per-claim random salts also make it infeasible to guess undisclosed values by brute force, even when the value space is relatively small.

2.4 DIDComm v2 Messaging

DIDComm is a **message-oriented communication protocol** designed for decentralized identity systems [15]. While Decentralized Identifiers (DIDs) provide cryptographic identifiers and Verifiable Credentials (VCs) provide portable attestations, DIDComm focuses on secure interaction: it defines how parties exchange protocol messages with **confidentiality**, **integrity**, and (when required) **sender authentication** at the message layer. Importantly, DIDComm is **transport-agnostic**: the security properties are attached to each message, not delegated to the underlying channel.

This chapter introduces the DIDComm communication model, its message structure and threading primitives, the packaging and cryptographic modes used to protect messages, the routing model with mediators, and finally an overview of the security properties offered by DIDComm in real-world applications.

2.4.1 Communication model

At a high level, DIDComm targets interactions between entities that control DIDs and associated keys. In SSI-style deployments, these entities often correspond to a holder wallet, issuer services, verifier services, and optional routing components (mediators). DIDComm standardizes:

- a common plaintext message model (what application protocols exchange);
- packaging rules (how plaintext becomes signed and/or encrypted envelopes);

- a routing model for indirect delivery through mediators; and
- interoperability hooks (profiles, protocol identifiers, feature discovery).

A core design choice is **asynchronous delivery**: transports are used only to deliver a message, and semantic responses are sent as new DIDComm messages rather than being returned on the same transport connection. This is well aligned with AI agent-based workflows where steps in a protocol may be separated by user actions or offline periods.

2.4.2 Parties and roles

DIDComm interactions typically involve:

- **Sender and Recipient**: endpoints that pack/unpack messages using keys referenced by their DIDs;
- **Mediators**: optional intermediaries that provide store-and-forward delivery and can bridge network boundaries (e.g., mobile agents behind NATs). Routing through mediators is expressed through message wrapping rather than exposing plaintext routing metadata.

A general schematic of a DIDComm communication flow is provided in Fig. 2.7.

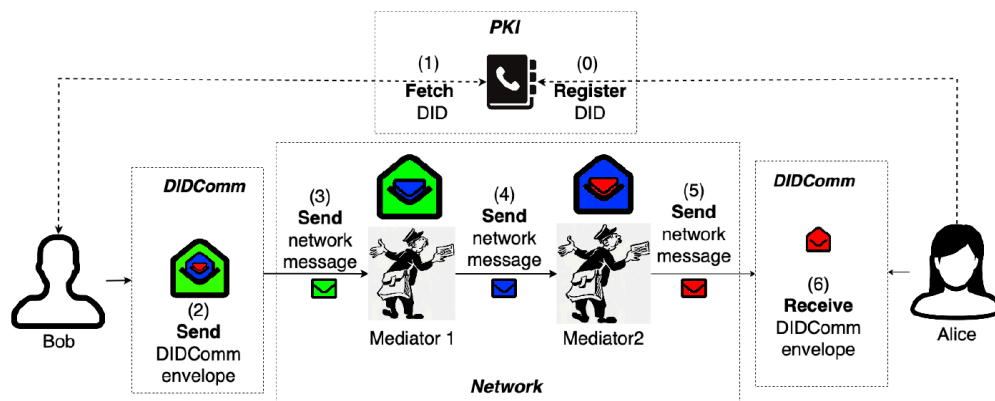


Figure 2.7: Overview of a DIDComm communication flow. Source: [16]

2.4.3 Message model

DIDComm v2 messages follow a common message model that stays consistent across protocols and transports. This section outlines the core building blocks of that model, deep-diving in how messages are structured, correlated across multi-step interactions, and extended with appropriate structures.

Plaintext Structure

DIDComm defines a standardized plaintext JSON structure, where most protocol-specific data lives in the **body**, and reusable metadata is expressed through headers. A typical plaintext message includes:

- **id**: message identifier;
- **type**: a message type URI that identifies the protocol message schema;
- **from** and **to**: sender and recipient identifiers (DIDs);
- **created_time** / **expires_time**: lifecycle hints;
- **body**: protocol-specific fields.

Plaintext headers (e.g., **id**, **type**, **from**, **to**, **thid**) and the **body** share the same protection boundary: when the plaintext message is packed, these fields are protected together and thus target the same intended audience.

Threading and Correlation

DIDComm provides explicit threading to correlate multi-step protocols. A thread is identified by **thid**; if absent in the first message, **id** is treated as the implicit thread identifier.

Messages in the same interaction include the same **thid**. When one interaction triggers another, a **pthid** header can link the child interaction to its parent thread. This supports composable protocols (e.g., a credential issuance flow spawning an auxiliary payment flow) without collapsing everything into a single state machine.

Acknowledgements

Since transports do not return semantic responses inline, DIDComm includes lightweight conventions for acknowledging messages. The sender can request an explicit acknowledgment using **please_ack**, and the recipient can respond using **ack** headers in the next messages. DIDComm includes conventions for reporting problems, which can be used to surface problems in asynchronous messages.

Attachments

Many identity protocols need to carry arbitrary payloads, e.g., JSON data, binaries, or referenced resources. DIDComm attachments offer a standardized way to include such data under an **attachments** header. They can be embedded as JSON data or base64url, or referenced as links with an integrity hash. They can also contain a

JWS to enable the signing of the content of the attachments separately from the content of the overall message.

This is useful in the VC space where large objects, e.g., credential offers, presentation definitions, or evidence documents, can be attached by value or by reference depending on the constraints.

Out-of-Band Messages

Out of band invitations are also supported by DIDComm. In this case, the invitation may be encoded as a URL or QR code. However, it should be noted to keep attention about putting sensitive information directly into these unencrypted messages, and identifiers revealed via out of band should be considered non-private and changed if privacy is desired.

2.4.4 Cryptographic packaging and message delivery

DIDComm distinguishes plaintext messages from their cryptographic envelopes. In practice, plaintext is typically packed into standard JOSE-based structures according to the DIDComm specification, using JWE for encryption and JWS for signatures.

Encryption Modes

DIDComm supports two core encryption styles:

- **Authenticated encryption:** also called **authcrypt**. The recipient can cryptographically verify the sender during decryption. The spec mandates the use of ECDH-1PU algorithm within a JWE structure for this mode. Authcrypt is the default choice when strong sender authentication to the recipient is required.
- **Anonymous encryption:** also known as **anoncrypt**. The message is encrypted for one or more recipients but without authenticating the sender to the recipients. It is suggested to use ECDH-ES under JWE for this modality. This mode of encryption might be required when the sender of the message is unknown in advance, but this impacts the privacy of the message.

Key Agreement and Message Signatures

DIDComm uses the recipient's **keyAgreement** keys, as defined in the DID Document (see Listing 2.1). These are a fundamental component of **confidentiality** in DIDComm, as they provide a sender with information about the types of cryptographic material that a recipient can use to compute shared secrets and decrypt

messages sent to them. This allows for true end-to-end encryption of messages while keeping transport and networking separate.

A DID may have multiple **keyAgreement** keys advertised under a single DID. This is particularly relevant in **multi-device scenarios** (e.g., a wallet on both a phone and a laptop), where a sender can encrypt a message under multiple recipient keys, and any authorized device can decrypt this message without needing a copy of the key. This design supports key rotation, device onboarding, and compartmentalization (each device can hold its own key material) while preserving interoperability.

DIDComm also supports **signed messages**. However, the spec notes that signatures are not required for tamper evidence or for proving the sender to the recipient when authenticated encryption is used; rather, signatures are useful when third-party verifiability (non-repudiation) is required, or when the recipient is not known at signing time.

Routing via Service Endpoints and Mediators

DIDComm relies on DID Documents to determine message delivery parameters. In particular, a DIDComm service entry may specify one or more service endpoints (transport URIs), the supported DIDComm profiles, and optional **routingKeys** that indicate mediator-based routing. Mediators are primarily useful for **store-and-forward** delivery and intermittent connectivity, as represented in Fig. 2.8.

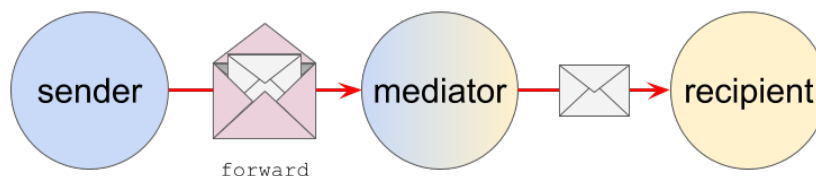


Figure 2.8: DIDComm message routing via a mediator. Source: [15]

When mediation is used, routing is realized through **message wrapping**: the sender encrypts the payload for the final recipient and encloses it in an encrypted **forward** message addressed to a mediator, which unwraps a single layer and relays the remaining ciphertext to the next hop. This design keeps routing metadata separate from the end-to-end protected plaintext (e.g., **to**), so intermediaries can forward messages without learning the inner content.

2.4.5 Security and privacy considerations

DIDComm provides message-layer confidentiality and integrity, and (depending on the mode) recipient-verifiable sender authenticity. However, the effective privacy of

a deployment depends on what the cryptographic layer inevitably reveals, what the underlying network reveals, and which operational choices (e.g., multi-recipient encryption, mediation, logging) are made.

Threat Model

DIDComm’s encryption protects the plaintext message (including DIDComm-level fields such as **id**, **type**, **from**, **to**, threading information, and the **body**) end-to-end between the communicating agents. Nonetheless, as with most secure messaging systems, some metadata can remain visible:

- **Network/transport metadata** (e.g., IP addresses, timing, and sizes) is not addressed by message-layer cryptography.
- **Envelope metadata** arises from the JOSE/JWE packaging used on the wire: some envelope headers must remain unencrypted to enable key selection and multi-recipient handling, even though they are integrity-protected.
- **Application metadata** may be intentionally revealed (e.g., acknowledgements, tracing, or protocol error reports), and should be governed by explicit policy.

A practical takeaway is that DIDComm offers **strong content protection**, but privacy-sensitive deployments must explicitly reason about who can observe which metadata and whether that metadata is acceptable for the intended use case.

Formal Analysis

Badertscher et al. [16] provide the first provable-security analysis of DIDComm v2 encryption modes, formalizing confidentiality, authenticity, and (sender) anonymity goals in a composable framework. Two conclusions are most relevant for system design.

Firstly, the analysis shows that DIDComm’s main modes can meet their expected guarantees (confidentiality and, in AuthCrypt, recipient-verifiable authenticity), yet they leak more than what an underlying network might leak. In particular, the current constructions reveal the **full recipient set** as part of the serialized ciphertext metadata, and in authenticated encryption this metadata can also enable sender linkage. This matters whenever the underlying network is expected to hide endpoints, for example in anonymizing networks. DIDComm’s cryptographic layer can re-introduce linkability by exposing who is talking to whom.

Secondly, the paper highlights that sequentially composing anonymous and authenticated encryption is not a great idea. By doing this, you don’t get the best of both worlds, but rather it can lead to unexpected behavior unless the

AEAD scheme [17] underlying the outer anonymous encryption layer is **compactly committing** [18].

Definition 1 (Compactly committing AEAD) *An AEAD scheme defined as $\Pi = (\text{Enc}, \text{Dec})$ is compactly committing if there exists a deterministic function $\text{Com} : \mathcal{A} \times \mathcal{C} \rightarrow \{0,1\}^\ell$ with $\ell = \text{poly}(\lambda)$ such that for any PPT adversary $\mathcal{A}$,*

$$\Pr \left[\text{Com}(a, \text{Enc}(k, n, a, m_0)) = \text{Com}(a, \text{Enc}(k, n, a, m_1)) \wedge m_0 \neq m_1 \right] \leq \text{negl}(\lambda),$$

where the probability is over $k \leftarrow \mathcal{K}$, $n \leftarrow \mathcal{N}$, $(a, m_0, m_1) \leftarrow \mathcal{A}(1^\lambda)$.

Intuitively, the ciphertext (together with the associated data) *commits* to a unique plaintext, in the sense that producing two different messages that yield the same compact commitment is computationally infeasible.

In the target architecture of this thesis, DIDComm is used primarily between known agents where the goal is confidentiality and recipient-verifiable authenticity (i.e., authcrypt-style operation), while strong anonymity against network observers is not a primary objective. Under these assumptions, the **recipient-set leakage** discussed above is typically **acceptable**: a network observer learning communication endpoints does not undermine the core security goals of the thesis.

Chapter 3

Agent2Agent and Model Context Protocols

Agentic systems increasingly operate as compositions of independently developed components. In practice, two integration directions dominate modern deployments. First, agents must collaborate with other agents to decompose complex goals into coordinated subtasks. Second, each agent must reliably access tools and data (e.g., enterprise APIs, databases, local resources) to ground decisions and execute actions. The lack of standard interfaces at either layer leads to framework-specific glue code, weak security boundaries, and limited portability across vendors and providers.

This chapter focuses on two emerging open standards that address these integration layers in a complementary manner. The **Agent2Agent (A2A) Protocol** standardizes *agent-to-agent* collaboration and task-oriented communication. The **Model Context Protocol (MCP)** standardizes *agent-to-tool/resource* connectivity and structured context exchange. This work presents their goals, core abstractions, interaction patterns, and security-relevant design choices. It then discusses how both protocols can be composed into an end-to-end multi-agent architecture.

3.1 Agent2Agent Protocol

A2A defines an open protocol designed to enable communication among “opaque” agentic applications, i.e., systems that can cooperate without revealing their internal mechanisms. A graphical overview of the A2A protocol is shown in Fig. 3.1.

The main goal is **interoperability** between different implementations: Agents built with different frameworks and platforms (such as LangGraph, CrewAI) should be able to communicate with one another using a shared language and operational semantics, despite having different architectures and assumptions. Interoperability

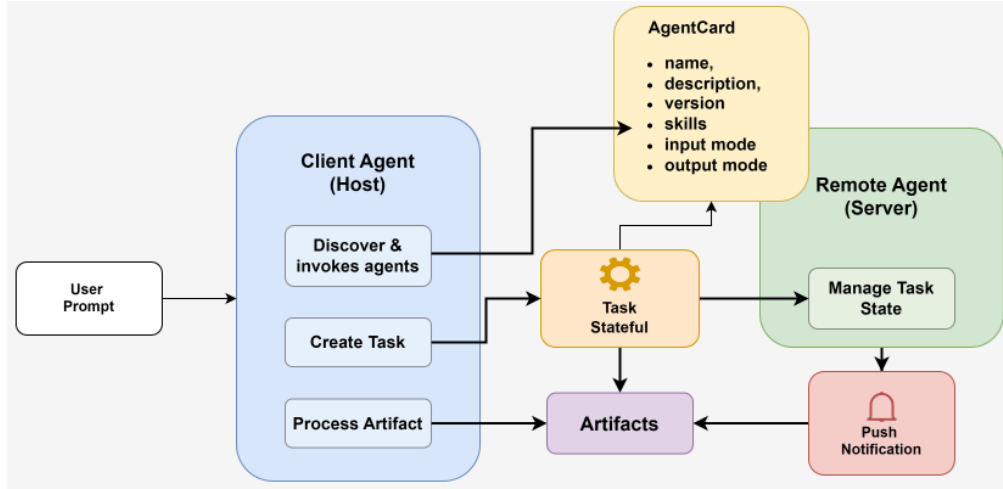


Figure 3.1: Overview of A2A Protocol core features. Source: [19]

is highlighted as one of the main goals in the official A2A documentation [20], along with the support for complex and multi-step workflows, as well as the black-box interaction paradigm where the agent does not necessarily need to reveal its proprietary information about the internal structure or decision logic.

From a technical point of view, A2A is defined as a **JSON-RPC-based protocol** [21], and it is supported by the use of standardized discovery metadata to advertise and negotiate capabilities of the agents. It also follows a **task-centric execution model**, in which the coordination is centered around the creation, delegation, and management of tasks.

In terms of roles, A2A typically distinguishes:

- **User:** the initiating principal (human or service) behind the goal.
- **Client Agent (A2A Client):** initiates requests on behalf of the user.
- **Remote Agent (A2A Server):** receives requests and executes tasks, possibly producing intermediate status updates and final artifacts.

These roles clarify responsibility for initiation, execution, and reporting, ensuring a structured flow from the initial prompt to the final output.

3.1.1 Discovery and Extensions

A2A adopts explicit **discovery** metadata through an **Agent Card**. An Agent Card functions as a machine-readable “business card” that describes an agent’s capabilities, endpoints, and authentication requirements. The A2A documentation positions the Agent Card as a cornerstone for discovery and capability negotiation

within an ecosystem. In typical deployments a client retrieves the Agent Card before invoking protocol methods, enabling the use of capabilities and/or authentication policies.

A2A is also design to be extensible. Indeed, **Extensions** offer a formalized way to introduce domain-specific negotiation logic, security mechanisms, or custom capabilities while maintaining interoperability expectations. From an engineering viewpoint, it is important that implementations treat extensions as first-class but optional modules: clients should only enable extensions offered by the Agent Card document, and servers should be agnostic to the absence or partial support of extensions.

3.1.2 Task-centric interaction flow

A2A is structured around the notion of a **Task** as the fundamental unit of work. Tasks have identifiers and **lifecycle states**, allowing interactions to span multiple turns and durations. The complete set of possible task states and the corresponding logical flow diagram are illustrated in Fig. 3.2.

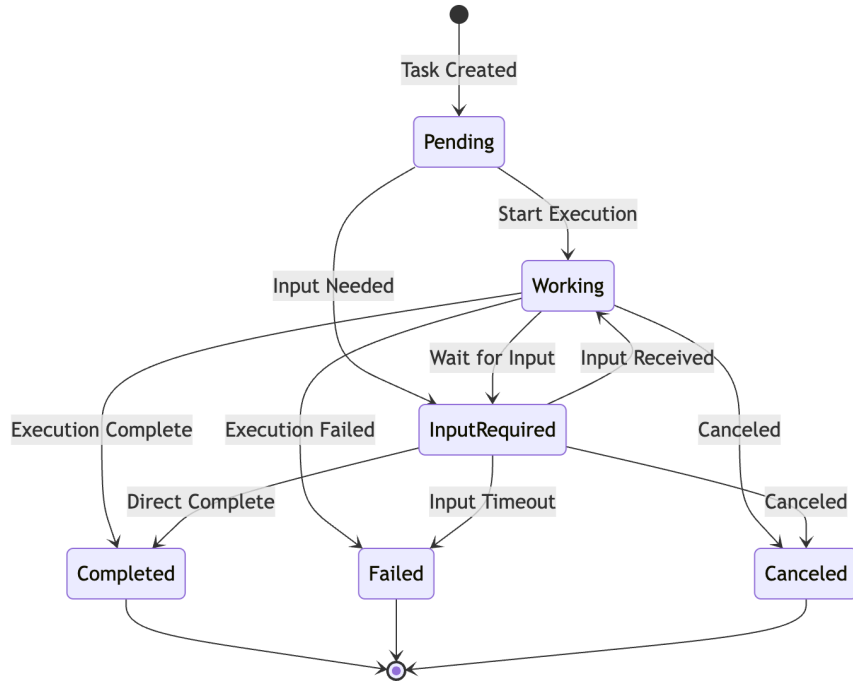


Figure 3.2: Task states. Source: [22]

The protocol facilitates both short cycle request/response and long-running tasks with asynchronous progress updates, referred to as **Notifications**. This lifecycle model is essential in multi-agent scenarios because it offers: **correlation**

of multiple messages and updates, **robust error handling** with terminal states, observability points (status transitions, artifact production), progressive interaction when additional input is needed. In this context, the Client Agent is a state interpreter because it needs to fetch and assess the status of the Task to identify the next feasible action.

A2A interactions exchange structured objects rather than untyped strings. The key communication element is the **Message**, which contains one or more **Parts**. Parts enable multimodality by representing text, structured data, and file payloads. The outputs of an agent are modeled as **Artifacts**, which can also hold one or more Parts. This architecture enables enterprise-style workflows where intermediate documents, structured outputs, or file attachments need to be passed through the protocol in a typed fashion.

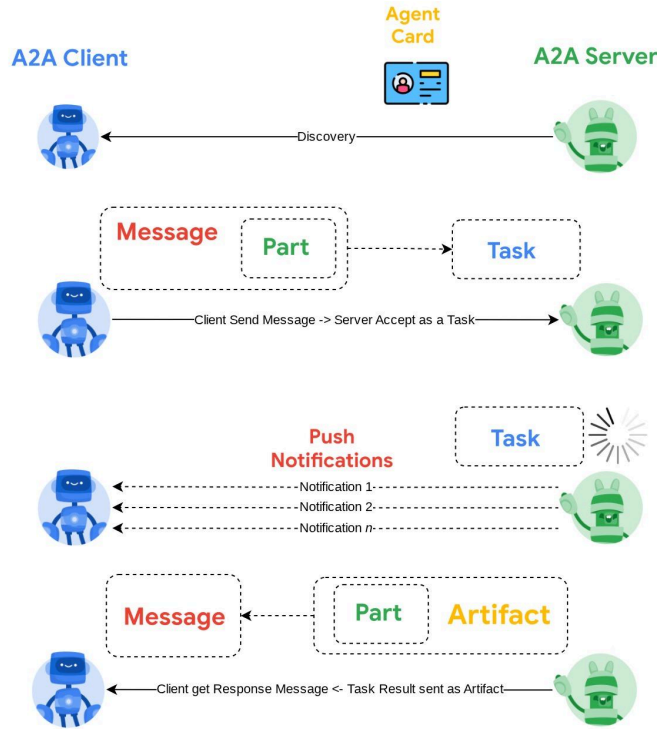


Figure 3.3: A2A example flow. Source: [23]

A2A supports both single-shot request/response interactions and incremental update delivery for long-running tasks. Figure 3.3 provides a schematic overview of a hypothetical **end-to-end A2A flow**: an A2A client discovers an A2A server via an Agent Card, submits a Task (composed by Message and Parts), and eventually receives a final Response, potentially including an Artifact.

In **synchronous** mode, the client waits for the final response, which fits short

tasks. For more complex executions, A2A can provide **streaming** of results using intermediate updates; this is illustrated in the "Push Notifications" sequence of Figure 3.3, where several notifications are sent by the server, each reporting progress and/or partial results of a task.

3.2 Model Context Protocol

MCP standardizes how AI applications connect to external systems. The official MCP documentation describes MCP as an open-source standard for connecting AI applications to data sources, tools, and workflows, frequently compared to a “USB-C” connector for AI integrations [24]. The protocol is intended to reduce integration complexity and make tool/data access portable across hosts and models.

3.2.1 MCP architecture

MCP adopts a client–server model, represented in Fig. 3.4, in which:

- an **MCP host** (often an AI application) manages user interaction and the overall session,
- an **MCP client** mediates protocol communication,
- an **MCP server** exposes capabilities (tools, resources, prompts) to the host/client

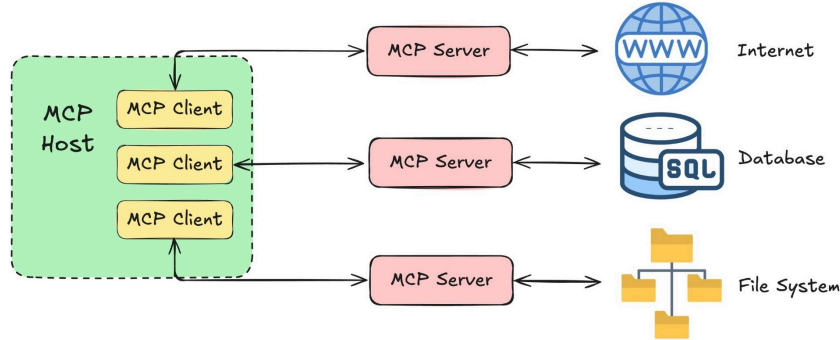


Figure 3.4: MCP client-server structure and example of capabilities. Source: [25]

This setup keeps the agent runtime separate from the capability provider, that is the component that exposes tools/resources. As a result, the same AI agent/host can connect to multiple MCP servers and dynamically choose the best tool or resource for the task at hand.

3.2.2 MCP server features

An MCP server exposes three complementary primitives to integrate models/agents with external capabilities in a clean and reusable way:

- **Tools:** represent executable actions. Tools have a distinct identifier and a structured data description (e.g., input constraints, output shape), allowing the client to discover the tools that are available, invoke the operation with structured parameters, and obtain a well-formed response (including error responses). This explicit interface helps reduce ambiguity and supports validation at the system boundaries, improving robustness and safety.
- **Resources:** provide retrievable context (e.g., documents, records, or other read-only information).
- **Prompts:** reusable instruction templates. Prompts allow an MCP server to offer standardized prompt templates that can be automatically detected and parametrized by the clients. This improves repeatability and reducing ad hoc prompt engineering across integrations.

3.3 A2A and MCP as a Composed Architecture

As stated in the documentation, these protocols are intended to be used in conjunction with one another, with the MCP protocol providing the agent to tool interaction layer, which works in conjunction with the A2A protocol, which manages the agent to agent collaboration at the higher level. Moreover, recent works specifically address the issue of the joint implementation of the MCP and the A2A protocols, including the implications and the design guidelines for the joint use of the protocols in real multi-agent systems [26] [27] [28].

3.3.1 Horizontal vs vertical integration

A helpful conceptual distinction introduced in [29] distinguishes A2A as **horizontal integration**, namely coordination between equal agents, and MCP as **vertical integration**, that is the use of tools and external resources to upgrade capabilities. This way of thinking about the two protocols is also reflected in comparative analyses of A2A and MCP, which emphasize that the two protocols aim at different levels of an agentic stack.

In Fig. 3.5 it's reported a schematic architectural view of a joint MCP-A2A deployment, adapted from [20], highlighting the division of responsibilities between the agent-to-tool interface (MCP) and the agent-to-agent coordination layer (A2A).

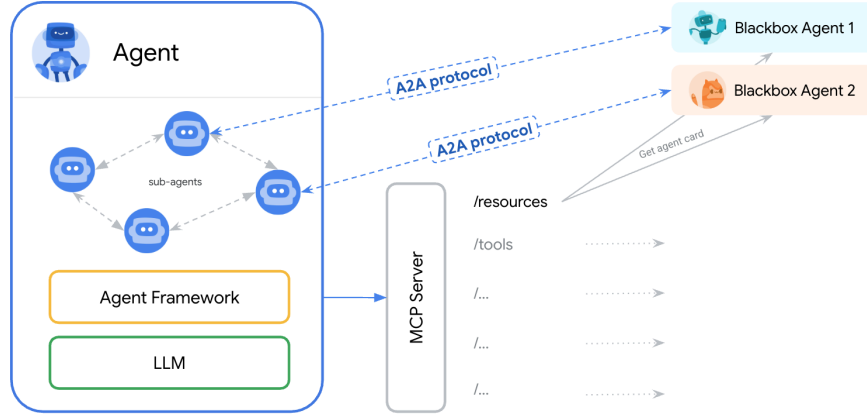


Figure 3.5: Example of integrated A2A–MCP architecture. Source: [20]

3.3.2 Bridging A2A coordination with MCP tooling

The main contribution of the composed architecture is **semantic encapsulation**: A2A exposes a task-oriented interface between agents, while the internal execution of a task may involve multiple MCP interactions that remain local to the executing agent. This design enables a coordinator agent to orchestrate a distributed workflow without requiring knowledge of how each specialized agent is implemented, which tools it uses, or how those tools are accessed.

Coordinator–specialist pattern

A common deployment pattern consists of a **coordinator agent** that receives the user goal and decomposes it into subtasks (e.g., information gathering, planning, booking, verification). Each subtask is delegated through A2A to a specialized agent. This organization is in alignment with the established multi-agent paradigm in which the achievement of tasks results from the structured dialogue between the involved agents with distinct roles, governed by a specific policy of orchestration [30]. The final response is then formed by aggregating the intermediate results.

Encapsulation of capability access

Within each delegated action, the agent executing the action may utilize MCP resources, call MCP tools, and even use MCP prompts as needed. Such interactions between the tools are generally confined to the agent and are not necessarily exposed to other agents unless the system is designed to do so (e.g., for debugging or transparency). This is in line with the overall idea of tool-based agents that combine reasoning with other actions such as APIs, while the high-level goal is

decoupled from the lower-level tool specifics [31]. Thus, the credentials are agent-local to the agent executing the action, and the tool layer is decoupled from the A2A logic.

Contract mapping at the boundary.

As A2A happens on the granularity of tasks and MCP happens on the granularity of tool call, the agent boundary has to relate the low-level outcomes of capabilities to the task-level semantics. This means ensuring a clear mapping between task progress and internal execution stages (including tool usage), transforming tool failures into task-level outcomes (such as retryable and non-retryable failures), and ensuring task outcomes are represented stably and typed (i.e., messages/artifacts) to be deterministically consumable by other agents downstream.

This mapping essentially enables the coordinator to be **agnostic** with regard to tools, while still reasoning about progress, partial results, and completion.

3.3.3 A2A and MCP sample skeletons

The listings below intentionally omit optional fields and extension-specific metadata: they serve only to show the shape of typical interactions and the point where A2A task execution can encapsulate MCP tool calls. Conceptually, the A2A server handling the delegated task can perform one or more MCP calls during execution.

A2A: message/send

```
1 {
2   "jsonrpc": "2.0",
3   "id": "req-001",
4   "method": "message/send",
5   "params": {
6     "message": {
7       "role": "user",
8       "messageId": "msg-001",
9       "parts": [
10        { "kind": "text",
11          "text": "Plan the itinerary and book hotels." }
12      ]
13    }
14  }
15 }
```

Listing 3.1: Example of message/send skeleton

MCP: tool discovery and invocation

```

1 { "jsonrpc": "2.0", "id": 1, "method": "tools/list", "params": {}
2   }
3 { "jsonrpc": "2.0", "id": 2, "method": "tools/call",
4   "params": { "name": "hotel.book", "arguments": { "city": "Rome"
5     } }
  }

```

Listing 3.2: Example of MCP tool calls

The details presented above are for illustrative purposes only; the normative requirements remain those defined by the respective specifications [20, 24].

3.3.4 Comparative summary

Table 3.1 summarizes key differences relevant to A2A protocol and MCP.

Dimension	A2A	MCP
Primary purpose	Agent-to-agent collaboration and task delegation	Agent-to-tool/resource connectivity and context exchange
Interaction pattern	Client agent $\leftrightarrow$ remote agent; task lifecycle and updates	Host/client $\leftrightarrow$ MCP server; tools/resources/prompts
Discovery	Agent Cards advertise skills/endpoints/auth	Capability listing (e.g., tools/list) and schema-driven interfaces
Outputs	Messages and artifacts (multimodal parts)	Tool results, resource payloads, prompt templates
System role	Horizontal interoperability layer in agent networks	Vertical capability layer for tool ecosystems

Table 3.1: High-level comparison of A2A and MCP

3.4 Transporting DIDComm Envelopes over A2A

In this thesis, A2A provides the orchestration layer for agent-to-agent coordination, while DIDComm provides a secure payload layer for exchanging cryptographic

protocol messages. This separation between these two protocols is very useful because it isolates concerns, indeed:

- **A2A** manages task lifecycle, delegation, artifacts, and higher-level coordination logic;
- **DIDComm** packages sensitive protocol payloads (e.g., presentation requests/responses, and error reports) with end-to-end cryptographic guarantees independent of transport.

A practical integration strategy is to treat a **DIDComm envelope** (typically in the form of `application/didcomm-encrypted+json`) as an **opaque payload transported through A2A**. In the prototype, this is implemented by embedding the DIDComm JWE object as a `DataPart` within an A2A message, explicitly tagging it with a `format` field obtained from configuration.

Outbound path (A2A → DIDComm request). When an agent needs to send a DIDComm protocol message, a possibility is to wrap a DIDComm JWE into a standard A2A `message.send` JSON-RPC request. Concretely, construct an A2A Message where:

- the DIDComm payload is placed in `parts` as `data.didcomm = {format, jwe}`;
- orchestration identifiers (`task_id` and `context_id`) can be attached to the A2A message, when available;
- optional `metadata` is attached at the A2A layer in `params.metadata`;
- a `json_rpc_id` is provided to correlate the JSON-RPC call/response at the transport/API level (distinct from task correlation)

Inbound path (A2A ← DIDComm response). On the receiving side, a possible integration is to represent the DIDComm response as an A2A `Task` carrying a dedicated artifact, namely creating:

- a `Task` with an explicit `TaskStatus.state` (e.g., `completed/pending`);
- an `Artifact` whose `parts` embed the DIDComm envelope as `data.didcomm = {format, jwe}`;
- optional `task_id` and `context_id` to preserve A2A-level correlation;
- optional `task metadata` for application-level bookkeeping.

This design keeps the DIDComm JWE opaque to A2A while still leveraging A2A artifacts as the delivery vehicle for the secure payload.

3.4.1 Correlation strategy

Both layers provide correlation mechanisms: DIDComm uses `id/thid/pthid` for protocol correlation, while A2A uses `task_id` and `context_id` for orchestration. A proposed robust design could be to carry both:

- **A2A correlation (orchestration):** `task_id` and `context_id` are explicitly propagated in outbound messages and reconstructed in inbound response tasks whenever available.
- **DIDComm correlation (protocol thread):** DIDComm thread identifiers are contained within the encrypted DIDComm message and, therefore, remain independent from the identifiers used at the transport layer.

This dual correlation strategy eliminates the risk of cross-talk between the orchestration and cryptographic protocol state. The A2A correlation can be used to manage the lifecycle and routing of tasks (`task_id/context_id`), and the DIDComm correlation can be used to manage end-to-end protocol continuity (via `thid/pthid`) without exposing sensitive content to intermediate layers.

3.5 Security Considerations in A2A–MCP Systems

A deployment model that makes use of both A2A coordination and MCP tool access is a multi-protocol execution chain. A2A is used for coordination between peers, while MCP is used for an agent host to connect with tools, resources, and prompts. This dichotomy is important for modularity but increases the attack surface since a vulnerability in one can be exploited to get to the other, with security assumptions silently breaking at the integration boundaries [29].

3.5.1 Threat surface and trust boundaries

An A2A–MCP system should be modeled as at least three trust zones: the A2A communication plane (discovery, task lifecycle, messages, artifacts), the MCP execution plane (tool selection and invocation, resource reads, prompt retrieval), and the external services ultimately affected by tool calls (databases, SaaS APIs, payment endpoints, identity systems). The most common failures occur when data, identities, or credentials cross zones without explicit constraints.

Two classes of inputs deserve special care because they are both machine-actionable and often natural-language adjacent:

- **Discovery metadata:** AgentCards in A2A and tool definitions in MCP are consumed to decide *who to trust* and *what to call*. Malicious metadata

can trigger social-engineering-style misrouting or instruction injection that influences an LLM-based planner.

- **Untrusted content payloads:** A2A message parts and artifacts, as well as MCP tool outputs, can carry text that an agent might treat as instructions. The system must prevent “data becoming control” across protocol boundaries.

3.5.2 Authentication and authorization

In many A2A applications, Bearer Tokens, such as OAuth2 access tokens incorporated or referred to within the task execution flow, may be used. The primary risk in an A2A–MCP pipeline is **credential transitivity**: a token valid for one zone that unintentionally enables actions in another zone, or remains valid far longer than it intended.

As reported by Louck et al. in [32], a robust strategy is to treat every boundary crossing as a re-authorization event:

- **Token audience binding:** tokens received by the A2A peer must not be valid at the MCP server, and vice versa. Ensure explicit enforcement of the token’s `aud` or audience, rejecting tokens not issued for the correct component.
- **Ephemeral, Task-scoped credentials:** issue tokens with shorter lifetimes, where the token scope is restricted to the task step or, better, the single tool call. This minimizes the attack blast radius of token leakage.
- **Least-privilege scopes with contextual limits:** where feasible, design scopes that are not too permissive, such as *"read everything"*. Instead, aim for scopes that contain the required data or functionality currently needed, such as the availability of calendars instead of the full calendars, or the cap on the amount to pay instead of the ability to make payments in general.
- **Anti-replay at the protocol layer:** add nonces/timestamps to sensitive task transitions and tool invocations; along with the requirement for idempotence keys where the operation must not be executed twice.

These controls are most effective when they are not merely “best practices” but enforced as part of the workflow state machine, so that sensitive steps cannot progress without the right authorization context.

3.5.3 Tool hardening

The MCP feature set provides a powerful capability plane for tools to read local files, make remote API calls, and execute actions with real-world effects. In the

context of MCP accessed via A2A delegation, the system must assume the caller (another agent) to be semi-trusted, and the content of the A2A to be adversarial.

Hardening measures should therefore focus on provenance, isolation, and policy enforcement:

- **Provenance and integrity of servers/tools:** only load MCP servers from known origins; pin versions; audit code; and prefer signed or otherwise verifiable tool manifests where available.
- **Runtime isolation:** run MCP servers with strong sandboxing (containers, restricted filesystem mounts, constrained network egress). If the tool is compromised, it should not be considered a host compromise.
- **Traffic mediation:** place MCP traffic through a gateway to enforce allowlists, rate limits, and inspection of requests and responses. This is especially important for tools with access to credentials and internal services.
- **Secrets management:** API keys and tokens should never be embedded within prompts or returned within tool output; they should be stored securely and retrieval should be narrowly scoped to specific operations.
- **Guardrails and policy checks:** before tool execution, evaluate a policy that considers the task context, the requesting agent identity, the tool risk level, and the target resource. Block or require extra approval for anomalous combinations.

These recommendations align with the MCP Security initiative from Cloud Security Alliance Community Project [33]. Indeed, the major threats reported are prompt manipulation, tool/metadata attacks, data exfiltration and protocol-level issues, keeping also an eye at isolation, traffic mediation, and operational controls.

3.5.4 Sensitive data handling

In multi-agent systems, keeping data minimization is not only a privacy principle: it also helps to mitigate the risks of prompt injection, accidental disclosure, and leakage between agents. In A2A–MCP chains, sensitive data will appear in the agent’s internal reasoning context, in messages exchanged between agents, and in the parameters of tool calls, so the system architecture should not put secrets in these channels unless absolutely necessary.

High-risk operations like payments, identity verification, or account connections should be conditional on explicit user consent, which is then stored as auditable metadata. Where possible, the most sensitive inputs should directly come from the user to the destination service, with agents merely conveying references or outcomes

instead of secrets. If personal data must be shared, sharing should be targeted and limited to what the verifier actually needs, without transmitting unnecessary context that might reveal other attributes. Overall, these measures reduce both inadvertent leaks and adversarial extraction through malicious peers or injected instructions [32].

3.5.5 Governance controls

The instrumenting of the A2A–MCP deployment needs to be viewed as an end-to-end governance process, not as two separate subsystems. This requires correlating the identifiers throughout the entire chain, from the A2A task and message identifiers to the MCP request identifiers. It also requires protecting the log data using mechanisms such as tamper-evident storage or digital signatures, so that the data is not deleted after the incident. Operationally, the system needs to **monitor** for detection signals such as increases in tool calls, unusual combinations of tool calls, probing behavior characterized by repeated failures and subsequent success, and policy denial leading to alternative execution paths.

Finally, the governance process needs to incorporate **rapid response mechanisms** such as fast rotation of keys, revocation of tokens, disabling of certain MCP servers/tools, and quarantining of peers. This way, the incident response process can be quickly implemented at the point where the A2A coordination meets the MCP capability execution, which is where the trust assumption is likely to fail.

Chapter 4

Thesis Objectives

Decentralized Identifiers and Verifiable Credentials unlock portable and cryptographically verifiable identity and authorization proofs, minimizing central IdPs dependencies and facilitating privacy-oriented disclosures. Still, major issues in this area are highlighted in recent literature. For example, in [9], Mazzocca et al. explicitly suggest that «domains other than those of traditional SSI should also be explored». This is in line with the fact that decentralized identity management is not limited to external customers but can also extend to internal and cross-organization use cases, where digital entities such as AI Agents must authenticate and prove their identity and attributes [34].

The focus of this thesis is on independent Generative AI agents that operate across multiple environments and stacks, making standardized **interoperability** and **security** a requirement to avoid specific or ad hoc integrations. Furthermore, it is essential to prevent impersonation, unauthorized access and data leakage issues.

For existing Agent-to-Agent systems, transport-layer HTTP Credentials such as Bearer Tokens and API Keys are commonly employed, with some systems utilizing also mutual TLS. These solutions are centralized in nature and, therefore, introduce centralized trust and potential single points failure. Moreover, authorization is often coarse-grained and difficult to express in a verifiable least privilege manner.

Building on recent work that motivates authenticated delegation as a core requirement for accountable AI agents [35], this thesis proposes the development of a modular security component with DIDs for agent identity, VCs for decentralized and granular authorization assertions, and DIDComm for secure and interoperable communication.

Therefore, the following enumerated thesis objectives are defined to connect the above research gaps to a concrete, end-to-end prototype in an agent setting.

Objective 1: Make VC flows usable for autonomous agents in a Microsoft infrastructure

Implement and adapt the Microsoft Entra Verified ID issuance/presentation flow so it can be used programmatically in an agent pipeline, without assuming an interactive mobile wallet as the sole operational model. The goal is to make a production-grade, cloud-deployable setup suitable for experiments and reproducibility, while remaining aligned with existing enterprise VC infrastructure [9] [34].

Objective 2: Improve privacy and reduce over-reliance on centralized Identity Providers

VC-based presentations can mitigate privacy risks typical of federated approaches such as correlation and excessive data exposure. By leveraging selective disclosure patterns and minimizing the information shared across agent interactions it's possible to reach this goal. The objective is to extend privacy-by-design principles discussed for wallets to agent-driven exchanges [9] [36].

Objective 3: Introduce decentralized authentication for A2A protocol

Design and integrate an authentication layer for Agent-to-Agent (A2A) interactions that is not limited to centralized identity assumptions. The objective is to enable agents to authenticate each other through decentralized identifiers and verifiable trust material, reducing impersonation risk and improving portability across administrative domains [9]. In support of this objective, this thesis introduces **a2a-didauth**, a purpose-built and reusable library that enables **DID-based mutual authentication** directly within A2A exchanges. The ideas behind this library were inspired by [37], with the ultimate goal to target general multi-agent AI deployments where autonomous AI agents must authenticate each other.

Objective 4: Express authorization and role separation via Verifiable Credentials

Define a credential-based authorization model where agent capabilities are represented as VC claims. This enables: explicit binding between an agent identity and its granted permissions, separation of roles through distinct credentials for different contexts. Finally, this method enables also least-privilege presentations since only the claims required by a verifier are disclosed.

This objective directly connects to the delegation problem for AI agents, where an agent should be able to act within a scoped mandate that is verifiable and auditable [9] [35].

Objective 5: Validate the composed A2A–MCP workflow under realistic threat assumptions

Demonstrate an end-to-end workflow where A2A coordination can safely delegate tasks to specialized agents that may internally invoke external tooling, while keeping authentication, authorization, and auditing properties coherent across the full pipeline. This includes designing failure tests such as crafted identities or credentials, replay attempts and misconfigured trust anchors to confirm the intended security properties [9] [35].

Chapter 5

Methodology and Architectural Design

This chapter describes the methodology used to create the architectural framework for the thesis. The main steps include the integration of decentralized identity management solutions with Microsoft Entra Verified ID and the creation of a custom wallet system that is accessible via API for automated credential management. At the end, this chapter proposed a simple and general overview on the end-to-end thesis prototype flow, to prepare the ground for the subsequent discussion.

5.1 Microsoft Entra Verified ID: Overview

Microsoft Entra Verified ID [38] provides an enterprise-grade verifiable credential service built on decentralized identity concepts. At a high level, it relies on Decentralized Identifiers to bind cryptographic keys to organizations and subjects, and on Verifiable Credentials to represent signed claims about a subject.

In this thesis, **Microsoft Entra Verified ID** is used as both **Issuer**, to generate organization-signed credentials, and **Verifier**, to request and validate presentations of those credentials.

5.1.1 Organization onboarding

To use Entra Verified ID in an application, the **tenant** must be configured so that the service can manage signing keys and publish the organizational DID information. For demonstrative and educational purposes, this thesis uses a dedicated Microsoft Entra test tenant created specifically for development, so that the Microsoft Entra Verified ID service could be enabled and used to issue and verify verifiable credentials in a controlled environment. This approach has been chosen in order to create a

free developer tenant intended for testing Verified ID scenarios, without impacting production identities.

After the tenant was created, the Verified ID service was enabled through the official tenant onboarding workflow in the **Microsoft Entra admin center**. As part of the setup, a fictitious organization was configured within the tenant to satisfy the prerequisites expected by the official documentation, at [38], and to proceed with the subsequent configuration steps and sample flows.

Later, in the advanced setup, the following three key steps were performed:

1. **Configure Azure Key Vault:** the Verified ID service that stores public/private key material in Key Vault. These keys are later used to sign and verify credentials.
2. **Register a Decentralized ID:** the tenant creates and registers its DID document to configure the trusted root.
3. **Verify domain ownership:** the DID is bound to an organizational domain so wallets can recognize and validate the issuer identity.

On the application side, an **Entra ID App registration** is required to get OAuth 2.0 access tokens when calling Entra Verified ID APIs for issuance or verification flows. In order to achieve this goal, the steps needed are the following:

1. register a new application in Microsoft Entra ID;
2. grant API permissions for the Verified ID Request Service, in particular flag the checkbox `VerifiableCredential.Create.All`;
3. grant admin consent, optionally separating issuance and presentation permissions across different applications if desired.

Operationally, this results in a standard enterprise deployment, even if this thesis is just a demonstrative test. The VC trust anchor (keys, issuer DID and domain binding) is managed by the Entra tenant, while custom Issuer/Verifier applications interact with the service through authenticated API calls.

5.2 Microsoft Entra Verified ID: Environment Setup

This section documents the concrete configuration used in the prototype, starting from the Microsoft Entra Verified ID tenant setup and ending with the creation of a custom credential contract used throughout the experiments. In the prototype, the following configuration refers both to the Issuer and the Verifier.

5.2.1 Tenant configuration

The Microsoft Entra Verified ID instance was created in a dedicated tenant and configured with:

- **Organization name:** org-test-did
- **Verified domain:** <https://stgacctstdid.z38.web.core.windows.net/>
- **License tier:** Microsoft Entra ID Free

The domain verification step is crucial because the issuer/verifier setup typically depends on publishing public artifacts under a controlled domain: a well-known configuration files or static resources referenced by credential manifests. In this thesis deployment, these public artifacts are hosted on an **Azure Storage static container** that is publicly reachable: the issuer DID is a `did:web` identifier, hence its DID document is resolved via HTTPS from the verified domain and served as a static JSON resource.

Concretely, the issuer/verifier DID document, listed in Lst. 5.1, is published under the verified domain cited above and exposes several useful fields.

```

1 {
2   "id": "did:web:stgacctstdid.z38.web.core.windows.net",
3   "@context": ["https://www.w3.org/ns/did/v1",
4     {"@base": "did:web:stgacctstdid.z38.web.core.
5     windows.net"}],
6   "service": [
7     {"id": "#linkeddomains", "type": "LinkedDomains",
8       "serviceEndpoint": {"origins": ["https://stgacctstdid.z38.
9       web.core.windows.net/"]}},
10    {"id": "#hub", "type": "IdentityHub",
11      "serviceEndpoint": {"instances": ["https://hub.did.msidentity
12      .com/v1.0/..."]}}
13  ],
14  "verificationMethod": [
15    {"id": "#...vcSigningKey-dd5f9",
16      "controller": "did:web:stgacctstdid.z38.web.core.windows.net",
17      "type": "JsonWebKey2020",
18      "publicKeyJwk": {"crv": "P-256", "kty": "EC", "x": "...", "y": "..."}},
19  ],
20  "authentication": ["#...vcSigningKey-dd5f9"],
21  "assertionMethod": ["#...vcSigningKey-dd5f9"]
22 }
```

Listing 5.1: Verifier/Issuer DID document

In particular, the `LinkedDomains` service binds the DID to the verified origin, the `IdentityHub` endpoint managed by the Microsoft infrastructure and the public JWK used for credential signing. Furthermore, Microsoft Entra Verified ID credential definitions include URLs, for example the logo URI, that must be publicly reachable.

5.2.2 Credential contract

A custom credential contract named `AdminCredential` was created as a lightweight test credential for the system integration. In Microsoft Entra Verified ID, a credential definition is described by two JSON documents:

- a **Display definition**, which describes how a wallet should render the credential as a card by defining colors, branding etc.;
- a **Rules definition**, which describes how claims are obtained and mapped into the resulting verifiable credential, and which credential types are declared.

Intuitively, display definitions focus on the front-end visualization layer, while rules definitions encode issuance requirements and entail how credentials claims are populated.

Display definition

```
1 {  
2   ...,  
3   "claims": [  
4     {  
5       "claim": "vc.credentialSubject.admin",  
6       "description": "true/false",  
7       "label": "Admin",  
8       "type": "Boolean"  
9     }  
10  ]  
11 }
```

Listing 5.2: Display definition for `AdminCredential`

The display definition is a locale-aware JSON object that controls the **card visual representation** in the Microsoft Authenticator wallet. For the sake of brevity and for better visualization, Lst. 5.2 includes only the portion relating to the credential's claims.



Figure 5.1: Visual representation of the AdminCredential

Rules definition

```
1 {
2   "attestations": {
3     "idTokenHints": [
4       {
5         "mapping": [
6           {
7             "outputClaim": "admin",
8             "required": true,
9             "inputClaim": "admin",
10            "indexed": true
11          }
12        ],
13        "required": true
14      }
15    ]
16  },
17  "validityInterval": 2592000,
18  "vc": {
19    "type": [
20      "AdminCredential"
21    ]
22  }
23 }
```

Listing 5.3: Rules definition for AdminCredential

Instead, the rules definition specifies the **attestation mechanism** used to obtain claim values, how these values are mapped into the output VC, the credential validity interval, and the VC **type** values declared by the contract. In particular, in the mapping array:

- `inputClaim` identifies the source field to read from the attestation input.

- `outputClaim` determines the name of the claim inserted into the issued VC.
- `required` indicates whether a given mapping must be present for issuance to succeed.
- `indexed` marks a claim value as usable for **searching** and **revocation**. Indeed, Microsoft Entra Verified ID requires exactly one indexed claim to make issued credentials searchable; if no claim is indexed, credentials cannot be revoked via the standard revocation workflow. The indexed value is not stored in cleartext: the issuer stores only a **salted hash** of the indexed claim, therefore lookups work only via exact matches. In practice, the admin revokes a credential by searching for the target record using the indexed claim. After revocation, the Request Service marks the credential as **REVOKED** and any future presentation will fail, unless the verifier explicitly allows revoked credentials.

The `validityInterval` is set to 2592000 seconds that correspond to 30 days, which is appropriate for thesis purposes such as iterative and interactive testing.

5.2.3 Attestation method

The prototype uses the `idTokenHints` attestation because it provides the most automation-friendly issuance model for a testbed with multiple autonomous agents.

Microsoft describes the **ID token hint** flow as an issuance mode where, cited directly from [38], «the relying party app needs to provide claims that should be included in the verifiable credential in the Request Service API issuance request». Then, it's the issuer application's responsibility to ensure that the required claims are included and correctly retrieved. For example, always citing the official documentation, the claims «can come from the current sign-in session, from backend CRM systems or even from self asserted user input».

However, this automation-oriented model exposes a number of **limitations** that need to be made explicit, particularly in autonomous and distributed deployments. First, the trust anchor for claim correctness is delegated to the issuer-side request service: the VC can be cryptographically valid, so properly signed by the issuer, even if the embedded attributes are weakly grounded. This happens because the protocol does not inherently enforce a verifiable provenance for the values injected via `idTokenHints`. As a consequence, the issuer application becomes part of the trusted computing base for the credential semantics. To keep it simple, if the backend is not properly configured or simply fed with unverified data, the resulting VC may transport inaccurate assertions while still passing signature checks at verification time.

Second, `idTokenHints` does not natively provide a mechanism to bind the issuance request to a pre-determined holder identifier (e.g., a specific DID) at the

protocol level. The issuance material is usually delivered as a QR code or deep link and it acts as a bearer artifact: whoever has access to it may try to redeem it, unless additional mitigations are put in place, such as short expiration windows and out-of-band PIN distribution. This last property, of course, is not necessarily problematic for interactive, user-centric applications. However, the situation gets more complicated for agentic applications, especially in terms of triggering issuance programmatically with eventual private attribute information.

In light of these constraints, the prototype intentionally adopts a **simplified assumption**: credentials are pre-issued and distributed to agents in a similar hardcoded manner. Essentially, the issuer app will be responsible for filling in pre-required claims via a static configuration and then issue one credential per agents identity, with a focus on the end-to-end feasibility of the issuance and consumption flow rather than on the realism of an enterprise-ready attribute sourcing. Hence, the topic of the advanced features of dynamic policying, attribute provenance, automated onboarding/offboarding, credential rotation, and fine-grained issuance authorization is considered out of the scope of the thesis and was left for future work.

To conclude, `idTokenHints` is a convenient way to drive issuance orchestration, but it does not by itself address authorization problems. In the meantime, `idTokenHints` is useful in a research prototype because it removes most of the operational overhead: since the claim population stays under the control of the issuer’s application, credential issuance flow can be automated, scripted, and repeated across multiple agents quickly and with ease.

5.3 Digital Wallet Walt.id

Many real-world issuer/verifier applications still deliver issuance and presentation requests via a QR code intended to be scanned in a mobile wallet, like the Microsoft Authenticator app. This is an effective UX for humans, but it becomes a **bottleneck** when the holder is an automated AI agent that must store credentials into a wallet without having access to manual scanning. Similarly, the AI agent should have the ability to present those credentials during verification flows. Therefore, the key methodology choice in this thesis is to treat the “wallet” as an **API-accessible subsystem** rather than a human UI application.

From a conceptual standpoint, the goal is to replace Microsoft’s proprietary wallet, i.e., Microsoft Authenticator, with the **open-source walt.id wallet** [39]. This enables fully automated issuance and verification flows, making the reference environment suitable for AI agents. Figure 5.2 provides a schematic overview of this idea, adapting a scheme from the official documentation.

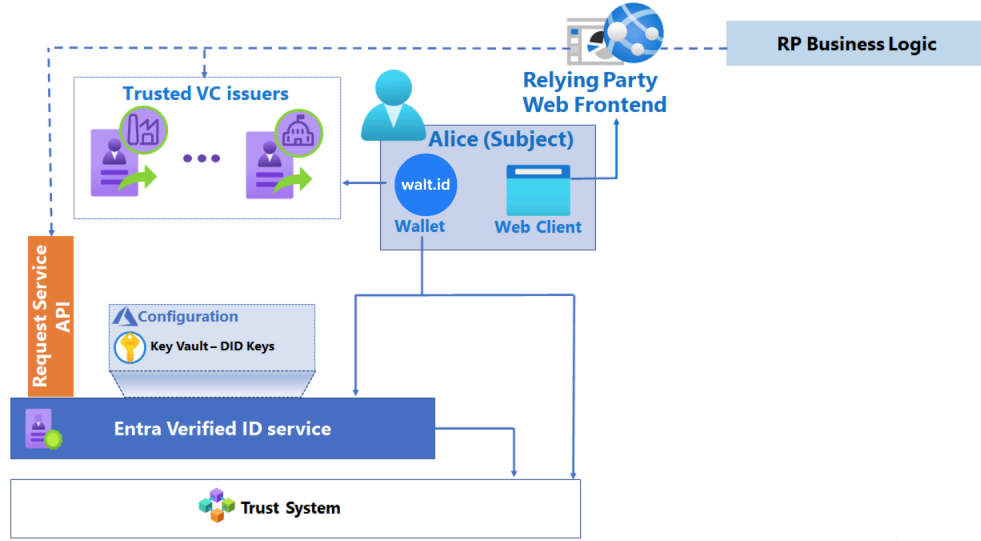


Figure 5.2: Schematic representation of walt.id wallet role. Source: [38]

Walt.id is an **API Wallet** specifically designed to provide endpoints for managing keys, DIDs, and multiple credential formats, and it explicitly supports credential receipt and presentation using **OIDC4VC protocols**, in particular both OpenID for Verifiable Credential Issuance (OIDC4VCI) [40] and OpenID for Verifiable Credential Presentations (OIDC4VP) [41]. So, an important note has to be made. Since Microsoft Entra Verified ID already drives issuance and presentation through flows aligned with OpenID for Verifiable Credentials, replacing Microsoft Authenticator with the walt.id API wallet is a seamless solution.

Furthermore, also DIDComm v2 integrates well with this kind of work. Indeed, a common drawback of OIDC4VC-type flows has been the tendency for the flow to end after issuance/verification, without leaving a standardized mechanism for a secure channel post-flows. In particular, De Prisco et al. propose a lightweight, backward-compatible approach where a DIDComm mediator data is introduced near the end of the OID4VC interaction, enabling the parties to establish a DIDComm secure channel after the credential exchange concludes [42]. In practice, this allows issuers/verifiers to deliver notifications and lifecycle events through DIDComm channel.

Hence, in practical terms, implementing the walt.id wallet provides three important benefits for the thesis system design:

- **Automation:** with walt.id requests that would normally be showed as QR codes to be scanned, can be processed automatically by the proper endpoints.
- **Isolation:** each agent can be mapped to a proprietary isolated wallet context,

reducing cross-agent key or credential leakage risk.

- **Separation of concerns:** agents keep their reasoning and task logic independent from credential storage details. Indeed, wallet operations are delegated to a dedicated service, made available to the agent via a MCP server.

The walt.id Community Stack can be run locally via **Docker**, via the *Compose* interface, including issuer, verifier, and wallet services plus a reference web applications. This makes it suitable for rapid prototyping and reproducibility in an experimental thesis environment. However, this thesis does not adopt the full stack: only the wallet component is replaced, since Microsoft Entra Verified ID remains the enterprise trust anchor that acts as both Issuer and Verifier.

5.4 Webhook Backend for Callbacks

During the implementation and testing of Microsoft Entra Verified ID issuance and presentation flows, the main questions were **observability** and **automation**. It's worth to note that Entra Verified ID workflows are inherently **asynchronous**: that means an Issuer/Verifier triggers a flow, the AI agent completes the interaction out-of-band (in the thesis case by making the wallet consume the deep-link) and only later the platform produces the final outcome through a **callback**.

A callback is simply an HTTP POST request that the Entra ID Request Service API sends to a predefined endpoint to notify the application of an event and deliver the associated payload asynchronously. In practice, this process creates a gap between the component that starts the process (i.e., the Issuer/Verifier) and the component that learns the final result (i.e., the endpoint receiving the callback log).

Without a dedicated callback receiver under my control, validating the flows became unhandy. Indeed, results had to be verified manually relying on untrustworthy sources such as UI feedback or ad-hoc debugging. Moreover, producing experiments and comparing behaviors across test runs is difficult, making integrating automated tests in agent-driven pipelines impractical. In other words, the real problem is find a way to turn an asynchronous, UI-oriented callback mechanism into a deterministic and machine-consumable interface.

5.4.1 Design and implementation

To solve the issues mentioned above, it has been developed a small backend whose purpose is to act as a **webhook endpoint** for Entra Verified ID callbacks and as a queryable event store for the thesis experiments. The implemented solution is available in the codebase at [43] on GitHub.

The backend was designed to ensure **traceability** of every issuance/presentation run via the `request_id` field, provide automation-friendly access to callback payloads through a simple HTTP GET, and extract relevant fields from `verifiedCredentialsData` such as `expirationDate` for programmatic checks inside the codebase.

At a high level, the backend implements a minimal workflow:

1. receive callback events from Entra Verified ID via HTTP POST;
2. the callback payload is stored and indexed by a stable identifier, namely the `request_id`, to enable reliable retrieval and following analysis;
3. expose a retrieval endpoint so that any component (tests, agents, scripts) can fetch the stored event later via a simple HTTP GET.

From an operational point of view, since the callbacks to the Verified ID service must be made to a publicly reachable HTTPS URL, the endpoints used for local development cannot be used for end-to-end testing. Therefore, in order to obtain a publicly accessible endpoint for the backend, the backend with the webhook has been deployed to **Microsoft Azure**. The URL that can be used to reach the webhook is:

`https://callback-webhook.azurewebsites.net/`

Deploying on Azure solves multiple practical issues at once. In particular, it provides a stable DNS name and TLS termination, ensures availability during repeated test runs, removes dependency on local reverse proxies and, above all, simplify the integration of the whole thesis architecture.

5.4.2 Service functionalities

The webhook service is a small **FastAPI**-based HTTP backend and, practically, it acts as both a public callback endpoint and a **lightweight event store** that can be accessed by automated tests, scripts or AI agents. From a functional perspective, the webhook exposes a minimal set of endpoints that address several needs:

- `GET /`: health check endpoint, used for monitoring and “wake-up” in scripts.
- `POST /`: callback ingestion endpoint. The request body is expected to be JSON and typically contains: `requestId` (required), and optionally could include `requestStatus`, `state`, `subject`, and `verifiedCredentialsData`.
- `GET /events/latest`: returns the latest events (by default the last 10), ordered from newest to oldest, useful for quick debugging during development.

- GET `/events/{request_id}`: returns all events associated with a specific `request_id`, ordered by timestamp.
- DELETE `/purge`: deletes all stored events (destructive; intended for test resets).

Callback data is stored in a **SQLite database**, keeping the service self-contained and easy to deploy. Each row includes both indexing fields and optional callback attributes. To keep retrieval efficient under repeated experiment runs, an index is defined over (`request_id`, `timestamp`). Given a known `request_id`, the retrieval endpoint returns the stored event payloads. When Verified ID data is present, the payload can include `verifiedCredentialsData`, from which the client-side tooling extracts `verifiedCredentialsData.expirationDate` to perform, for example, checks on validity windows.

As a final result of this discussion, the webhook backend serves as a thin but important integration component, which takes the callback feature of the platform and turns it into a reliable and queryable interface for experiments, debugging, and evaluation without relying on manual inspection of portals and logs.

5.5 Architectural Concept

An overview of the conceptual architecture of this thesis is presented in Fig. 5.3. The system is composed of the following building blocks:

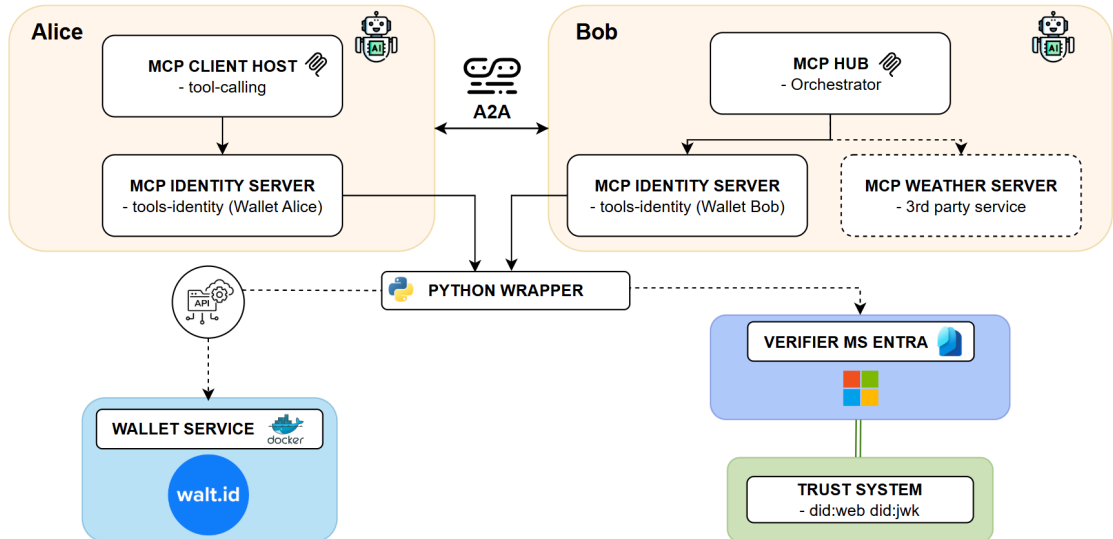


Figure 5.3: High-level architecture of the thesis prototype

- **AI agents:** autonomous agents coordinating through the A2A protocol. In the proposed model, there are only two of them: Alice and Bob, but in theory, there could be any number of them in a network with many components. Each of the autonomous agents plays the roles of subject and holder, with **one DID** and **an independent wallet**, to isolate secrets and credentials.
- **DIDComm v2 layer:** used on top of A2A messaging to provide end-to-end confidentiality, integrity, and sender authentication between AI agents. It's a further layer of protection for end-to-end communications.
- **MCP servers:** expose external capabilities as tools, with two main categories:
 - **Identity MCP servers** which provide credential-related operations by exploiting walt.id wallet APIs;
 - **Third-party MCP servers:** included to show that credential-protected agent workflows coexist with ordinary tool use. In particular, this server is available on GitHub [44] and provides an MCP-compatible weather service that exposes forecasting capabilities through structured tool calls. In practice, this server allows an agent to request and retrieve current conditions or forecasts for a given location.
- **Stateless Python wrapper:** an adaptation layer that translates agent-level intents (e.g., “store credential”, “build presentation”) into concrete wallet API calls.
- **Wallet walt.id:** the programmable wallet back-end used by agents to manage DIDs, keys and handle credential exchange.
- **Microsoft Entra Verified ID:** the enterprise issuer/verifier providing credential trust anchoring (keys in Key Vault, DID registration, domain binding and API-driven issuance/verification).

5.6 End-to-end Workflow

To avoid repeating background in the following chapters and to already have some sense of clarity, this section summarizes the high-level operational workflow overview implemented in the prototype. The goal is to show how credential issuance, agent authentication, and request authorization are merged into a single end-to-end interaction (Alice → Bob), using a weather query as the demo example.

1. **Verifiable Credential issuing.** An issuer application initiates the issuance flow with Microsoft Entra Verified ID, so that a target AI agent can obtain a Verifiable Credential containing the proper claims. To achieve one of the

objectives of this thesis, the agent-side wallet is automated: the issuance request is consumed through a deep-link and completed algorithmically thanks to walt.id endpoints, without a human-in-the-loop UI.

2. **Webhook backend deployment for callbacks.** Issuance and, similarly, verification follows an asynchronous pattern: the relying service triggers the flow and later receives the final outcome through a callback mechanism. For this reason, the prototype includes a dedicated webhook backend that collects these callbacks and exposes the resulting state to the agents; its design and implementation are described in details in the next chapter.
3. **Mutual authentication between agents via a2a-didauth.** Before delegating actions, Alice and Bob authenticate each other at the application layer through the `a2a-didauth` library, a custom component developed in this thesis and representing its main innovation. The two parties prove control of their respective DIDs via a challenge/response exchange carried inside A2A messages, so the peer identity is clear and established cryptographically.
4. **Authorization via Verifiable Presentation and access grant.** When Alice attempts to invoke a protected operation, Bob requires evidence of authorization. Alice constructs a Verifiable Presentation derived from her Verifiable Credential submits it to the Microsoft Entra Verified ID verifier endpoint. The latter validates the VP (e.g., by checking integrity, issuer trust and required claims) and, if successful, then Bob (A2A Server) concedes the authorization on its protected resource by granting a short-lived access token that permits to the caller (Alice) to unlock them.
5. **Demo over DIDComm: weather request Alice → Bob.** After the token is obtained, Alice issues a high-level request such as «get the weather for a given location». This interaction happens over the the DIDComm secure channel established after authentication and authorization phase. In the thesis workflow, DIDComm is not treated as a generic chat transport: it is a post-authorization secure channel used to carry the application request (weather query) and the corresponding result.

Bob retrieves weather information by invoking the relevant weather tool [44], using the MCP tool layer, and returns the result to Alice on the same DIDComm channel, so that both the protected task execution and its outputs remain bound to the verified agent identities.

Chapter 6

System Implementation

This chapter addresses the main technical aspects required to implement the conceptual architecture introduced in Section 5.5. Specifically, the proof-of-concept setup is described by examining the role of each component in the prototype, from the A2A server surface and routing execution model to the VC-based authorization flow and the DIDComm-protected messaging. This discussion is implementation-oriented and provides the necessary context for Chapter 7, which later will presents the proposed framework for authenticating AI agents via DIDs. All references in this chapter are grounded in the project GitHub repository, available in [45].

6.1 AI Agents

The core of the implementation involves AI agents designed to handle the interaction between different subsystems. These agents are structured around an abstract class, named **Agent**, that serves as the blueprint for the various AI agents in the system. The agents inherit from this abstract class, which defines the key behaviors, specifically, how to connect to the MCP identity server and also how to setup the interfaces needed to resolve secrets and private keys.

Each agent has **one DID** and its VCs, moreover, they're programmed to handle specific tasks that involve communication with other agents, invoking APIs, or interacting with external resources. The use of an abstract class provides flexibility, as it allows for the easy addition of new agent types or modification of existing ones without altering the core functionality of the system.

6.1.1 Integration with walt.id wallet

One of the main principles of software development in the modern world is encapsulation of complex logic in software components. In the context of this project,

encapsulation of the logic of the wallet API of `walt.id` is of critical importance, in order to achieve a clean architecture and easy integration of other services. By encapsulating the interaction-logic of the Wallet APIs, this thesis aims to abstract complexity for the rest of the system too. In addition, encapsulation of the logic of the Wallet ID API in a single client ensures that if any changes are needed, they can be implemented in a single place without risking errors.

In particular, the `client.py` script, located in the `common/waltid_core` directory of the project repository, plays a crucial role in encapsulating the Wallet ID API. This script acts as a client that interacts with the Wallet ID API by providing an abstraction layer that simplifies the integration of the API into the overall project. As stated in Lst. 6.1, this file contains a class which aggregates all the `@classmethod` required to have access to the `walt.id` wallet. This includes tasks such as the authentication of the user, handling of tokens, and making the necessary API calls in the most secure and efficient way.

```
1 class WaltIdClient:
2     _client: Optional[httpx.AsyncClient] = None
3
4     @classmethod
5     async def authenticate(cls, email: str, password: str) ->
6         WaltIdSession:
7         ...
8
9     @classmethod
10    async def get_default_did(cls, session: WaltIdSession) ->
11        Optional[str]:
12        ...
13
14    @classmethod
15    async def get_credentials_list(cls, session: WaltIdSession) ->
16        list:
17        ...
```

Listing 6.1: `WaltIdClient` API call wrapper

This wrapper written in the file `client.py` is **stateless**, which means it does not retain any state between each of the successive function calls. This means that each call to the Wallet ID API will be independent of each other, ensuring that the client will not retain any information from previous calls. Moreover, the **modular structure** of this client has been identified as beneficial since it will be possible to use the client within multiple components of the proof of concept that will be utilizing the Wallet ID API. This will eliminate the possibility of duplicated code and, of course, improve maintainability and consistency throughout the proof of concept.

6.1.2 Integration with the Model Context Protocol

In the thesis proposed architecture, each agent will have both horizontal coordination through inter-agent communication using A2A, and vertical capability access using MCP. More specifically, each agent will function as an **MCP host** and, during execution of a task, will establish one or more Client Sessions towards dedicated MCP servers, discovering and invoking tools and resources as needed. This maintains separation of the agent runtime and specific integrations, as already previously mentioned in Chapter 3.

Next, a technical deep dive is provided for each of the agents in the conceptual architecture, Alice and Bob, regarding integration with the Model Context Protocol.

Alice: MCP Identity Server

Alice interacts with an **MCP Identity Server** which provides identity and wallet operations through MCP primitives. From the architectural perspective, this module functions like an "identity sidecar," where the agent functions as the MCP host, and the server contains all sensitive material and logic specific to the backend. In essence, the wallet operation logic simply reuses the `client.py` module from Subsection 6.1.1, reducing duplicated code and enhancing coherence.

Identity-related operations, such as handling DID key material, are sensitive and should not be handled by leaking private information into the context of the LLM. By concentrating this functionality through the MCP, the agent can make requests for tool calls for cryptographic operations while keeping sensitive materials, as private keys and tokens, isolated.

At startup, the agent spawns the MCP server as a local subprocess using **stdio transport** and immediately establishes an **MCP ClientSession**, as hinted in Lst. 6.2. In the prototype, this is done so that all subsequent wallet operations are performed through structured MCP tool calls rather than direct API invocations.

```
1 async def main(city: str):
2     alice = Alice()
3
4     await alice.mcp_connect(
5         command="uv",
6         args=["run", "python", "-m", "alice.mcp.waltid_server"]
7     )
```

Listing 6.2: MCP Identity Server spawn

To expose wallet functionality to the agent through a uniform MCP interface, the stateless `WaltIdClient` implemented in `client.py` is wrapped inside a `FastMCP` [46] server that re-exports its methods as MCP `@tool` endpoints. A visual example is provided in Lst. 6.3.

```

1 from mcp.server.fastmcp import FastMCP
2 from common.waltid_core import WaltIdClient as waltid
3
4 mcp = FastMCP("Alice-waltid-wallet")
5
6 @mcp.tool()
7 async def authenticate(email: str, password: str):
8     return await waltid.authenticate(email, password)
9
10 @mcp.tool()
11 async def get_default_did(session):
12     return await waltid.get_default_did(session)
13
14 def main() -> None:
15     mcp.run(transport="stdio")

```

Listing 6.3: MCP Identity server skeleton with FastMCP

Bob: MCP Hub

From the identity perspective, Bob follows the same pattern as Alice: wallet and credential operations are exposed through an MCP server, therefore there is no conceptual difference in how the identity server is started and accessed.

What changes in Bob is the introduction of an **MCP Hub** layer, implemented as a lightweight client-side aggregator that maintains multiple MCP `ClientSessions` and offers a unified interface to enumerate and invoke tools across different MCP servers.

```

1 async def mcp_connect(self, command: str, args: Optional[list[str]
2     ] = None) -> None:
3     # 1) WEATHER MCP server (3rd party)
4     await self.mcp_hub.add_stdio_server(
5         server_key="weather",
6         command=command,
7         args=["run", "mcp_weather_server"],
8     )
9
10    # 2) WALT.ID MCP server (custom)
11    await self.mcp_hub.add_stdio_server(
12        server_key="waltid",
13        command=command,
14        args=["run", "-m", "bob.mcp.waltid_server"],
15    )
16
17    await self.mcp_hub.refresh_tools()

```

Listing 6.4: Bob connection to the MCP Hub

In practice, Bob connects the Hub to a third-party **weather MCP server** [44] that provides real-time current conditions for a given location, exposed as callable tools the MCP Hub can invoke when answering weather-related requests. Moreover, the MCP Hub is also equipped with the custom **walt.id MCP server**, already discussed for Alice. The codified names are respectively `mcp_weather_server` and `bob.mcp.waltid_server`; and in Lst. 6.4 there's a snippet taken from the thesis GitHub repository that explains those concepts.

```

1 class McpHub:
2     def __init__(self) -> None:
3         self._stack = AsyncExitStack()
4         self._sessions: Dict[str, ClientSession] = {}
5         self._tool_map: Dict[str, ToolRef] = {}
6         self._openai_tools: List[dict] = []
7
8     async def add_stdio_server(self, *, server_key: str, command:
9 str, args: List[str]):
10         server_params = StdioServerParameters(command=command,
11 args=args)
12         read, write = await self._stack.enter_async_context(
13 stdio_client(server_params))
14         session = await self._stack.enter_async_context(
15 ClientSession(read, write))
16         await session.initialize()
17         self._sessions[server_key] = session
18
19     async def refresh_tools(self) -> None:
20         self._tool_map.clear()
21         self._openai_tools = []
22         for server_key, session in self._sessions.items():
23             resp = await session.list_tools()
24             for tool in resp.tools:
25                 alias = f"{server_key}.{tool.name}" # avoid
26 collisions
27                 self._tool_map[alias] = ToolRef(server_key, tool.
28 name)
29
30     async def call(self, tool_alias: str, arguments: Dict[str, Any
31 ]) -> Any:
32         ref = self._tool_map[tool_alias]
33         session = self._sessions[ref.server_key]
34         return await session.call_tool(name=ref.tool_name,
35 arguments=arguments)

```

Listing 6.5: MCP Hub skeleton

Operationally, each server is registered via `add_stdio_server`: the Hub spawns the target server using `StdioServerParameters`, opens the stdio streams via

`stdio_client`, instantiates an MCP `ClientSession`, and performs `initialize()` to negotiate capabilities. Once all sessions are active, `refresh_tools()` pulls `list_tools()` from each server and builds a collision-free tool namespace by prefixing every tool name with its `server_key` (e.g., `weather.get_current_weather`). In Lst. 6.5 the core structure of the `McpHub` is shown.

To sum up, the Hub choice is motivated by two concrete needs. In the next paragraph, the motivations that lead to this implementation will be revealed.

Bob: LLM deployment

In practice, the LLM backend is configured via environment variables, keeping secrets out of the GitHub codebase and enabling reproducible deployments across local and cloud environments. For the prototype, Bob is equipped with a GPT deployment named `gpt-5-mini` and uses the preview API version `2024-08-01-preview`.

```
1 export AZURE_OPENAI_API_KEY="***"  
2 export AZURE_OPENAI_ENDPOINT="https://<resource-name>.azure.com/"  
3 export AZURE_OPENAI_DEPLOYMENT_NAME="gpt-5-mini"  
4 export AZURE_OPENAI_API_VERSION="2024-08-01-preview"
```

Listing 6.6: Bob LLM configuration via environment variables

Circling back to the MCP Hub concept, it comes in handy mainly for two reasons:

- **tool aggregation:** there are multiple MCP servers but the agents, for simplicity, needs just only one interface,
- **governance of LLM-visible capabilities:** Bob can selectively expose only a subset of aggregated tools to the LLM (e.g., weather only), while keeping identity-critical tools (the waltid ones) callable exclusively from deterministic code paths.

The second motivation is very important because, in the prototype, Bob delegates the understanding of Alice’s human language and correlated response generation to an Azure OpenAI deployment. However, the LLM is not invoked as an unconstrained internal chatbot: it is integrated into Bob as the reasoning component that drives actions through MCP. Bob maintains an MCP client, via the MCP Hub, that aggregates the available tool surface and exposes it to the LLM only after applying an explicit allowlist. As a result, the model cannot directly access system resources or arbitrary network endpoints; instead, all external actions are performed exclusively by calling the MCP tools that Bob decides to publish, with clear namespacing and a bounded set of parameters, reducing the attack surface of prompt-influenced tool usage.

In the prototype, this goal is achieved by carefully specifying a set of `llm_tools_allowlist` entries identified by the server key, e.g. `"weather"`, while identity-critical tools such as `"waltid"` remain callable only from deterministic code paths. Whenever a tool failure occurs, such as timeouts, upstream errors, invalid parameters, the error is looped back to the LLM in a structured way, and Bob can either propagate the failure as an A2A Task error or fall back to a deterministic response policy.

6.2 DIDComm v2: Setup and Messaging

In the prototype, DIDComm v2 is used as a **message-layer security envelope** that protects the application payload end-to-end, independently from the orchestration transport. Concretely, Alice and Bob rely on the Python `didcomm` library [47] to **pack** plaintext messages into a JWE container and to **unpack** received JWEs back into plaintext.

The A2A layer is then used only as a carrier: the DIDComm JWE is embedded as an opaque artifact within an A2A `DataPart` preserving a clear separation between coordination and confidentiality, as already discussed in Section 3.4.

6.2.1 ResolversConfig: DID and Secrets resolution

DIDComm v2 does not require a persistent connection or a dedicated session handshake. In this thesis, a «channel» is considered ready when both peers can:

- resolve the other party’s DID into a `DIDDoc` suitable for DIDComm,
- provide the private key material required by the library during decryption.

```

1 def set_resolvers_config(self) -> None:
2     did_resolver = StaticDIDResolver()
3     secrets_resolver = SecretsResolver4JWK(
4         waltid=waltid,
5         session=self.waltid_session,
6         cache=True
7     )
8     self.resolvers_config = ResolversConfig(
9         secrets_resolver=secrets_resolver,
10        did_resolver=did_resolver,
11    )

```

Listing 6.7: Configuration of `ResolversConfig` for Agents

Therefore, the channel becomes effectively usable as soon as both parties configure a `ResolversConfig`; that means provide the library the capabilities to retrieve

the DID document given a decentralized identifier and the handling of private-key material. Therefore, both agents instantiates a static DID Resolver and a Secrets Resolver that delegates private-key retrieval to the walt.id wallet, enabling also optional caching for repeated operations.

The DID resolver manually builds a DIDComm-compatible DIDDoc from the output of the local DID resolution logic, pointing the attention to explicitly mapping verification material as JWK, as listed in Lst. 6.8.

```

1 class StaticDIDResolver(DIDResolver):
2     async def resolve(self, did: DID) -> DIDDoc:
3         doc = custom_did_resolver(did)
4         vm = VerificationMethod(
5             id=doc["verificationMethod"][0]["id"],
6             type=VerificationMethodType(1), # JSON_WEB_KEY_2020
7             controller=doc["verificationMethod"][0]["controller"],
8             verification_material=VerificationMaterial(
9                 format=VerificationMaterialFormat(1), # JWK
10                 value=json.dumps(doc["verificationMethod"][0]["
11                 publicKeyJwk"])
12             ),
13         )
14         return DIDDoc(
15             did=doc["id"],
16             key_agreement_kids=doc["keyAgreement"],
17             authentication_kids=doc["authentication"],
18             verification_methods=[vm],
19             didcomm_services=[],

```

Listing 6.8: Static DID resolver

```

1 class SecretsResolver4JWK:
2     async def get_key(self, kid: DID_URL) -> Optional[Secret]:
3         # ... parse did:jwk public part from kid ...
4         base_private = await self._get_base_private_jwk() #
5         exported from wallet
6         if not self._same_public_jwk(requested_pub, base_private):
7             return None
8         jwk_for_didcomm = dict(base_private)
9         jwk_for_didcomm["kid"] = kid # MUST match requested DID
10        URL (incl. "#0")
11        secret = jwk_to_secret(jwk_for_didcomm)
12        return secret

```

Listing 6.9: Secrets Resolver integrated with walt.id wallet

On Secrets Resolver side, in thesis repo, the `SecretsResolver4JWK` class exports the wallet default key and returns a DIDComm-compatible secret only if the

requested `kid` corresponds to the same embedded public JWK. A critical implementation detail is that the resolver must satisfy the exact DID URL requested by the library (including fragments such as `#0`). The snippet is shown in Lst. 6.9.

6.2.2 Message pack and unpack workflow

At the core of the `didcomm` Python library, two built-in functions implement the secure messaging lifecycle:

- **pack_encrypted**: takes a plaintext DIDComm message together with the configured resolvers and produces an encrypted DIDComm envelope (JWE) ready to be transported over any medium,
- **unpack**: consumes a received JWE, resolves the necessary DIDs and secrets, and reconstructs original plaintext message, including its metadata and body.

In the remainder of this section, these two operations are described in practical terms by following the two sides of the exchange in the prototype: first the Alice side, which packs and sends a DIDComm request, and then the Bob side, which unpacks, processes the payload, and packs the DIDComm response.

Sender side: Alice

```

1 didcomm_msg = DidcommMessage(
2     id=str(uuid.uuid4()),
3     type="example/1.0/weather-request",
4     body={
5         "message": f"Ciao Bob, quale è il meteo a {city}?",
6         "access_token": access_token
7     },
8     frm=sender_did,
9     to=[BOB_DID],
10 )
11
12 pack_result = await pack_encrypted(
13     resolvers_config=resolvers_cfg,
14     message=didcomm_msg,
15     frm=sender_did,
16     to=BOB_DID,
17     sign_frm=None,
18     pack_config=PackEncryptedConfig(protect_sender_id=False,
19                                     forward=False),
20 )
jwe_request_json = json.loads(pack_result.packed_msg)

```

Listing 6.10: Crafting of a JWE DIDcomm message with `pack_encrypted`

The application payload is encoded as a DIDComm message whose body contains both the user query and the short-lived authorization token obtained after the VC-based authorization step. In the project repo, the helper function `build_didcomm_weather_request` builds a DIDComm message and packs it to a JWE via `pack_encrypted` built-in function. This way of proceeding is shown in Lst. 6.10.

Then, Alice embeds the packed JWE as an opaque payload into an A2A message/send JSON-RPC request by wrapping it inside a `DataPart` under the `didcomm` container, as reported in Lst. 6.11.

```

1 data = {"didcomm": {"format": DIDCOMM_FORMAT, "jwe":
2     didcomm_jwe_req}}
3 msg = Message(
4     message_id=str(uuid.uuid4()),
5     role=Role.user,
6     parts=[Part(root=DataPart(data=data))],
7 )
8
9 json_rpc = SendMessageRequest(
10     id=json_rpc_id,
11     params=MessageSendParams(message=msg, metadata=metadata),
12 )
13
14 a2a_didcomm_request = json_rpc.model_dump(mode="json",
15     exclude_none=True)

```

Listing 6.11: Embedding DIDComm inside A2A

In the end-to-end flow, Alice sends this JSON-RPC payload to Bob's A2A endpoint and uses the same `json_rpc_id` as correlation handle to validate the response before unpacking it.

Receiver side: Bob

On reception, Bob first validates that the incoming A2A message contains a DIDComm container and extracts the JWE payload. Then it calls `unpack` to recover the plaintext DIDComm message and reads the `message` and `access_token` fields from the message body. Authorization is enforced at this stage (before any LLM or tool execution), and the response is packed again as a DIDComm JWE and returned to Alice as an A2A Task artifact. All these phases are sum up in code in Lst. 6.12.

Finally, the executor wraps the DIDComm reply JWE into an A2A Task artifact using the same `data.didcomm={format,jwe}` structure, and enqueues it as the JSON-RPC response payload, see Lst. 6.13.

```

1 # validate A2A structure and extract JWE
2 jwe_str = _validate_and_get_jwe(a2a_msg)
3
4 # unpack DIDComm
5 unpack_result = await unpack(self.resolvers_cfg, jwe_str)
6 user_msg = unpack_result.message.body.get("message")
7 token = unpack_result.message.body.get("access_token")
8
9 # authorization gate
10 response_content = await self.
    _authorize_and_process_weather_request(token, user_msg)
11
12 # build DIDComm reply and pack to JWE
13 reply = DidcommMessage(
14     id=str(uuid.uuid4()),
15     type="example/1.0/weather-response",
16     body=response_content,
17     frm=self.did,
18     to=[ALICE_DID],
19 )
20 pack_result = await pack_encrypted(
21     resolvers_config=self.resolvers_cfg,
22     message=reply,
23     frm=self.did,
24     to=ALICE_DID,
25     sign_frm=None,
26     pack_config=PackEncryptedConfig(...),
27 )
28 didcomm_response_jwe = json.loads(pack_result.packed_msg)

```

Listing 6.12: Bob operations on receiver side

```

1 data = {"didcomm": {"format": ..., "jwe": <didcomm_resp>}}
2
3 didcomm_artifact = Artifact(
4     artifact_id=str(uuid.uuid4()),
5     name="didcomm_response",
6     parts=[Part(root=DataPart(data=data))],
7 )
8 task = Task(
9     id=task_id,
10    context_id=context_id,
11    status=TaskStatus(state=task_state),
12    artifacts=[didcomm_artifact],
13    metadata=metadata,
14 )

```

Listing 6.13: Wrapping the DIDComm response as an A2A Task artifact

6.3 A2A Agent Card Design and Authorization Mechanisms

The agents are intended to communicate in a bidirectional manner, managing the flow of tasks within the system. The agents communicate with each other using the A2A (Agent-to-Agent) communication protocol, adopting the JSON-RPC 2.0 transport model and its task-oriented interaction pattern. In this setting, **Alice** acts as an **A2A client**; while **Bob** exposes an **A2A server** endpoint over HTTPS, publishing an Agent Card for discovery and advertising both capabilities and authentication requirements.

However, first of all, it's worth to say that, since the A2A specification mandates HTTPS in production deployments, modern TLS configurations has been used in the prototype. Indeed, agent Bob makes use of Uvicorn [48] to run its A2A server with a self-signed TLS certificate. On the other side, agent Alice is configured to trust the TLS certificate, ensuring encryption-in-transit and server identity verification within the experimental setup.

6.3.1 Discovery via Agent Card

```

1 def create_agent_card(url: str, did: str, authenticated: bool =
  False) -> AgentCard:
2     didauth_skill = AgentSkill(
3         id="didauth",
4         name="DID Authentication (DIDAuth)",
5         description="...",
6         security=None, # public
7     )
8     weather_skill = AgentSkill(
9         id="weather_didcomm",
10        name="Weather over DIDComm",
11        description="Return current answer for weather requests",
12        security=[{"bearer": []}] if authenticated else None
13    )
14    card = AgentCard(
15        name="Bob Weather Agent",
16        url=url,
17        protocol_version="0.3.0",
18        preferred_transport="JSONRPC",
19        skills=[didauth_skill] if not authenticated else [
20        didauth_skill, weather_skill],
21        supports_authenticated_extended_card=True,
22    )

```

Listing 6.14: Bob Agent Card construction

In A2A, agent discovery is driven by the **Agent Card**, a JSON document that describes the agent identity, supported transport, skills, and declared security schemes. The protocol recommends serving the card at a well-known location (e.g., `/.well-known/agent-card.json`) and optionally supporting an **Authenticated Extended Card** for revealing additional protected capabilities after login.

Specifically, in the prototype, Bob publishes:

- a **public Agent Card**, exposing just the prerequisite information to complete the DID-based mutual authentication flow (commonly referred as `a2a-didauth` in this thesis),
- an **Authenticated Extended Agent Card**, which additionally exposes the protected **weather skill** guarded by a Bearer/JWT scheme.

This split makes the prerequisite explicit at discovery time, while keeping protected skills hidden until authorization is completed. In Lst. 6.14 the snippet presents this idea, showing in details how Bob’s Agent Card is constructed.

6.3.2 Authenticated Extended Agent Card

To effectively protect the authenticated card retrieval endpoint, i.e., the method `agent/getAuthenticatedExtendedCard`, and the related capability disclosure, the A2A Server treats `agent/getAuthenticatedExtendedCard` as a **protected method**. In practice, a Starlette middleware inspects incoming JSON-RPC requests and enforces authentication only when this specific method is invoked, while keeping the rest of the A2A surface accessible without a token.

```

1 class ProtectExtendedCardMiddleware(BaseHTTPMiddleware):
2     async def dispatch(self, request: Request, call_next):
3         if path in self.rpc_paths and request.method.upper() == "
4         POST":
5             payload = json.loads(body.decode("utf-8") if body else
6             "{}")
7             methods = {payload.get("method")} if isinstance(
8             payload, dict) else set(...)
9             if any(m in self.protected_methods for m in methods):
10                 if not self._has_valid_bearer(request):
11                     return JSONResponse({"error": "Unauthorized"},
12                     status_code=401)
13                 return await call_next(...)

```

Listing 6.15: Middleware enforcing Bearer token for `agent/getAuthenticatedExtendedCard`

Access to the protected method is unlocked by attaching a Bearer token: this token is granted only after the caller successfully presents a valid Verifiable Credential to Bob, so that the authenticated card can be retrieved exclusively by an authenticated and authorized agent. In Lst. 6.15, the implementation of the extended card protection mechanism is illustrated.

Instead, in Fig. A.3, the sequence diagram shows how Alice retrieves an authenticated extended AgentCard from Bob. After establishing an A2A TLS channel, Alice submits a Verifiable Presentation to the Microsoft Entra ID Verifier, then, if the outcome is positive, Bob emits a short-lived JWT authorization token that Alice uses to request the extended card.

6.4 Router-Based A2A Execution Model

From the A2A perspective, Bob exposes a single HTTPS endpoint, but internally it supports multiple interaction paths. To avoid confusion between protocol-specific state machines inside one giant monolithic handler, in the prototype an **executor split approach** is adopted. Indeed, a **Router Executor** acts as the only A2A entrypoint, while concrete protocol handlers are implemented as dedicated executors. This design keeps the external A2A surface stable, keeping the same endpoint and task semantics while assuring clean and simple separation of concerns within the server.

6.4.1 Router Agent Executor

At runtime, the A2A server forwards each incoming request to just a single **AgentExecutor**. In this thesis specific case, this is the **RouterAgentExecutor**, whose only responsibility is to inspect the **A2A RequestContext** and delegate execution to the correct internal handler. Concretely, in the thesis prototype, two executors implement the protocol-specific logic:

- a **A2ADidAuthExecutor** for the **a2a-didauth** authentication-oriented messages
- a **DIDCommExecutor** responsible to manage the **DIDComm** channel and all the application flow, after the authentication phase between the AI agents is completed

From a practical point of view, the router therefore behaves like an internal demultiplexer, ensuring that each executor remains small, testable, and strictly focused on one flow.

The router identifies the intended flow using observable A2A-level cues, without performing any cryptographic operation itself. In particular, it detects:

- a `a2a-didauth` interaction by checking extension activation headers and extension-scoped metadata: `metadata[ext_uri].op`;
- an application message by locating a `DataPart` carrying the expected payload structure: a non-empty envelope under `data.didcomm`).

Lst. 6.16 reports the two functions responsible to discriminate to which flow the incoming `RequestContext` is intended.

```

1 def is_didauth_flow(self, context: RequestContext) -> bool:
2     if not helpers.is_didauth_ext_activated(context, ext_uri=self.
3         _ext_uri):
4         return False
5     meta = helpers.get_metadata(context)
6     ext_meta = meta.get(self._ext_uri)
7     return isinstance(ext_meta, dict) and ext_meta.get("op") in ("
8         begin", "reject", "response")
9
10 def is_didcomm_message(context: RequestContext) -> bool:
11     for part in context.message.parts:
12         root = getattr(part, "root", None)
13         if isinstance(root, DataPart):
14             didcomm = (root.data or {}).get("didcomm", {})
15             return didcomm.get("format") == DIDCOMM_FORMAT and
16                 bool(didcomm.get("jwe"))
17     return False

```

Listing 6.16: Flow-discrimination functions

Once the router determines the intended flow, it forwards the exactly same `RequestContext` to the corresponding executor. This pattern maintains a single entrypoint, i.e. the Router, while keeping protocol logic encapsulated behind a shared executor interface. This methodology is shown in Lst. 6.17, where each executor is called upon receiving the proper `RequestContext`.

```

1 async def execute(self, context: RequestContext) -> Task:
2     if self.is_didauth_flow(context):
3         return await self._didauth_executor.execute(context)
4     if self.is_didcomm_message(context):
5         return await self._didcomm_executor.execute(context)
6
7     # default: reject unexpected payloads deterministically
8     return helpers.error_task(
9         context=context,
10         message="Unsupported message format for Bob endpoint"
11     )

```

Listing 6.17: Delegation to the specialized executors

Overall, the router-based split helps to provide a clean implementation: it preserves a single, stable A2A endpoint while separating protocol-specific responsibilities into dedicated executors. As a result, DIDAuth and application handling can evolve independently, without forcing changes in the `RouterAgentExecutor` interface. Furthermore, authentication-related parsing or state transitions remain confined to their own component rather than leaking into the application logic.

From an operational standpoint, Bob simply wires the router as the unique `AgentExecutor` of the A2A server, so every incoming `message/send` request is deterministically dispatched to the appropriate internal executor, permitting to the external task lifecycle and semantics at the A2A layer to remain unmodified.

6.5 Issuer Backend Application

In the thesis prototype, an **Issuer application** is required because issuing a Verifiable Credential with Microsoft Entra Verified ID is a tenant responsibility: a holder wallet cannot build a credential on its own.

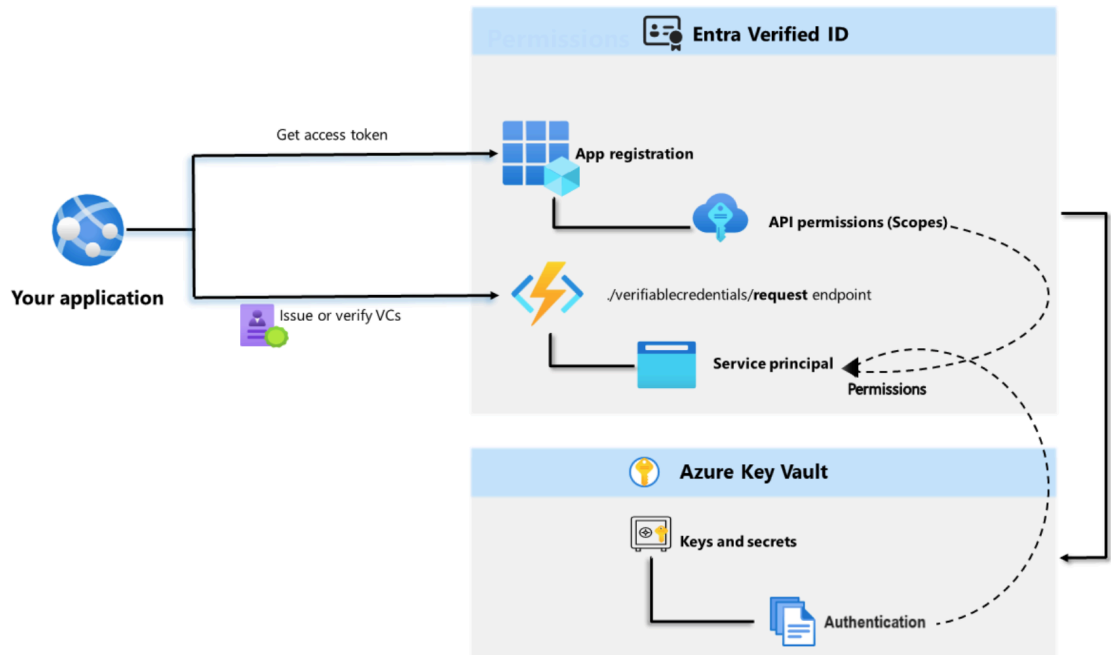


Figure 6.1: Issuer application overview. Source: [38]

As shown in Fig. 6.1, a trusted backend must authenticate as an Entra application, obtain an OAuth2 access token with the correct scopes, and call the Request Service API to create an issuance session. Only after this step a wallet-consumable

`request_uri` exists (i.e, a deep-link or a QR in interactive deployments), and the holder can complete the issuance. As already discussed, the issuer app also communicate with a callback endpoint so that the Verified ID service can asynchronously notify the progress of the issuance by means of two specific events: `request retrieved` and `issuance successful`.

6.5.1 Access token acquisition

The issuer app first requests an OAuth2 access token from the Microsoft identity platform v2 token endpoint using `grant_type=client_credentials`. The returned Bearer token is then used to authorize the issuance-session creation call.

```
1 def get_oauth_token(tenant_id, client_id, client_secret, scope):
2     url = f"https://login.microsoftonline.com/{tenant_id}/oauth2/
3         v2.0/token"
4
5     data = {
6         "client_id": client_id,
7         "client_secret": client_secret,
8         "grant_type": "client_credentials",
9         "scope": scope, # typically ".../.default"
10    }
11
12    headers = {
13        "Content-Type": "application/x-www-form-urlencoded"
14    }
15
16    resp = requests.post(
17        url,
18        data=data,
19        headers=headers
20    )
```

Listing 6.18: OAuth2 `client_credentials` against the v2 token endpoint

6.5.2 Issuance request creation

With a valid `access_token`, the issuer app calls the Verified ID Request Service API endpoint configured as `ISSUER_API_URL`. The payload encodes the issuance configuration: the authority DID, the `manifest`, the credential `type` (for the prototype it is `AdminCredential`), the `claims` to embed (`admin=true/false`), and the callback settings. The callback block includes a correlation `state` and an API key header; the webhook backend uses this header to authenticate incoming status notifications.

```

1 def vc_issue(endpoint, callback_url, did, manifest, oauth_token,
2   claims):
3     headers = {
4         "Authorization": f"Bearer {oauth_token}",
5         "Content-Type": "application/json",
6         "Accept": "application/json",
7     }
8     payload = {
9         "includeQRCode": False,
10        "callback": {
11            "url": callback_url,
12            "state": str(uuid.uuid4()),
13            "headers": {"api-key": "callback-api-key-SPIKEREPLY2025"}
14        },
15        "authority": did,
16        "registration": {"clientName": "Microsoft Entra Verified
17        ID"},
18        "type": "AdminCredential",
19        "manifest": manifest,
20        "claims": claims
21    }
22    resp = requests.post(url=endpoint, headers=headers, data=json.
23    dumps(payload), timeout=30)
24    resp.raise_for_status()
25    return resp.json()
26
27 resp = vc_issue(
28     endpoint=ISSUER_API_URL,
29     callback_url=CALLBACK_URL,
30     did=DID_AUTHORITY,
31     manifest=MANIFEST,
32     oauth_token=get_oauth_token(TENANT_ID, CLIENT_ID, OAUTH_SECRET
33     , OAUTH_SCOPE),
34     claims={"admin": "true"} # "true" or "false"
35 )

```

Listing 6.19: Creation of an issuance request

After the code in Lst. 6.19 is executed, the response is the automation pivot: the issuer app obtains a server-side **requestId** and a wallet-consumable **request_uri** in the **openid-vc://** form. In a standard interactive deployments the same value would be encoded into a QR; but since the prototype targets **automation**, it is returned to the agent pipeline and fed directly into the holder wallet API.

6.5.3 OID4VCI issuance automation

On the holder side, the agent authenticates to the wallet service and submits the issuance `request_uri` to the exchange endpoint `/wallet-api/wallet/{wallet_id}/exchange/useOfferRequest`. This call programmatically replaces the scan and accept interaction: the wallet resolves the request, completes the issuance protocol with the Verified ID service, and stores the resulting Verifiable Credential.

```

1 requests.post(
2     BASE_URL + f"/wallet-api/wallet/{wallet_id}/exchange/
   useOfferRequest",
3     headers={"Authorization": f"Bearer {waltid_token}", "Content-
   Type": "text/plain"},
4     data=request_uri,
5     timeout=15
6 )

```

Listing 6.20: Issuance offer consumption via walt.id endpoint

Issuance progress is reported asynchronously by the Verified ID service to the callback URL configured in the issuance request. Furthermore, all callbacks events are generated after the code in Lst. 6.20 is executed.

Optionally, the AI agent can query the webhook backend via the endpoint `/events/{requestId}` to verify completion, that means find in the log an event of type `request_retrieved` followed by a `issuance_successful`. As an additional deterministic check, the holder wallet can list its stored credentials via the endpoint `/wallet-api/wallet/{wallet_id}/credentials`.

In Fig. A.1 there's the sequence diagram that makes explicit the concrete HTTP calls used in the implementation and shows how the digital walt.id wallet effectively replaces the human QR scan, thanks to the compatibility with OID4VCI standard.

To conclude, the introduction of an Issuer Application enforces tenant-level governance, while the walt.id wallet automates the holder-side completion by exchanging the `openid-vc://` offer request for a proper Verifiable Credentials. This dichotomy is similar to the intended operational model, where issuance is a creation of a trusted backend, while acceptance and storage remain holder-side responsibilities, even in the context of automated holder interaction as in this thesis context.

6.6 Verifier Backend Application

In the prototype, Bob includes a **Verifier backend** responsible for initiating the presentation flow with Microsoft Entra Verified ID. This component is crucial because it produces the presentation request that encodes the authorization policy

in terms of credential type, trusted issuer, and sets the constraints on the required claim. Lst. 6.21 shows a real implementation snippet, highlighting just the payload that Bobs sends to the Request Service Verifier API of MS Entra Verified ID.

```

1 payload = {
2   "authority": DID_AUTHORITY,
3   "callback": {
4     "url": CALLBACK_URL,
5     "state": state,
6     "headers": {"api-key": CALLBACK_API_KEY},
7   },
8   "registration": {"clientName": "MS Entra Verified ID"},
9   "includeReceipt": False,
10  "requestedCredentials": [
11    {
12      "type": "AdminCredential",
13      "purpose": "To verify that you have admin
14  authorization",
15      "acceptedIssuers": [DID_AUTHORITY],
16      "configuration": {
17        "validation": {
18          "allowRevoked": False,
19          "validateLinkedDomain": True
20        }
21      },
22      "constraints": [
23        {"claimName": "admin", "values": ["true"]}
24      ]
25    }
26  ]
27 }

```

Listing 6.21: Creation of a presentation request

Similarly to issuance, the verifier backend first obtains an OAuth2 access token via the v2 token endpoint, like in Lst. 6.18, and then calls the Verified ID verifier API endpoint.

Afterwards, as shown in the snippet above, the Verifiable Credentials that Alice presents must satisfy the Verifier policy embedded in the request: the wallet must produce a presentation that contains an `AdminCredential` issued by a trusted issuer (i.e., `DID_AUTHORITY` that corresponds to the `did:web` listed in Lst. 5.1) and that discloses a claim `admin=true` according to the declared `constraints`. In addition, Bob asks the Request Service to enforce that only non-revoked credentials are accepted by setting the `allowRevoked=false` field.

The response contains the server-side correlation identifier `requestId` and a wallet-consumable link. In interactive scenarios this is rendered as QR code; in the thesis prototype it is consumed programmatically by the walt.id wallet to

complete the OIDC4VP presentation without manual interaction. Finally, exactly like in the issuance case, the verification progress and final outcome are delivered asynchronously to the webhook backend configured in the requests callback block.

6.6.1 OIDC4VP presentation automation

Similarly to the issuance automation shown in Lst. 6.20, the prototype is also able to complete the OIDC4VP presentation process without a human-in-the-loop. After Bob creates a presentation request and obtains a wallet-consumable link, Alice forwards that link to the walt.id wallet exchange endpoint. A precise representation of this flow is shown in Fig. A.2.

A small but important note has to be made. In the prototype, the presentation is not executed through the standard walt.id “exchange” endpoints: instead, the OIDC4VP flow is implemented **manually** by consuming the `request_uri`, decoding the presentation-request JWT, selecting the Verifiable Credential(s) that satisfies the `presentation_definition`, and crafting the required artifacts. Concretely, the agent **locally signs** both the `vp_token` (a JWT-VP embedding the VC) and the self-issued `id_token` carrying the `presentation_submission`, then submits them to the Verifier’s `redirect_uri` via a POST requests. For a detailed view on the code, all this process is written in the official thesis repository on GitHub [45].

6.6.2 Admin access token grant

Once the presentation is verified, the system must translate the credential-level proof `AdminCredential` presented under the verifier policy into a simple authorization primitive consumable by A2A protocol. In the thesis prototype, this primitive is a short-lived **Bearer token** used to unlock the protected method `agent/getAuthenticatedExtendedCard`, and, consequently, the disclosure of protected capabilities.

Taking a step back, it is important to clarify the two roles that coexist in this thesis workflow:

- **Microsoft Entra Verified ID** acts as the **Verifier service** for the OIDC4VP presentation: it evaluates the presentation against the verifier policy and reports the outcome to Bob through the configured callback channel.
- **Bob** acts as the **authorization authority** for the A2A API surface: only after observing a successful verification outcome it issues the short-lived token used to access `agent/getAuthenticatedExtendedCard`.

From a security point of view, the Bearer admin token is treated as an **ephemeral capability**. It is binded to Bob’s A2A server, so it cannot be reused elsewhere;

and it has short lifetime (120 seconds) and a binding to an `expected_holder_did` in order to prevent the possibility of a replay attack.

Operationally, the Starlette middleware shown in Lst. 6.15 enforces the presence and validity of the Bearer token only for the protected method, leaving the rest of the A2A surface reachable to complete prerequisite steps. In other words, as already discussed, capability disclosure is gated: Bob reveals additional skills only after a successful authorization step, preventing unauthenticated callers from discovering or invoking protected operations.

Chapter 7

Validation

This chapter aims to give a full overview on the innovative parts of the thesis by explaining the characteristic flows of authentication of the AI agents using Decentralized Identifiers, and authorization via Verifiable Credentials. Moreover, the application flow will also be presented, a simple demo in which the agents fulfill the goal of, respectively, request and answer a weather related question. The validation approach used in this chapter is divided in two parts: firstly, a theoretical validation will be proposed explaining the intended goals and the security assumptions behind them. Finally, five Python tests will be presented, simulating operative attack and failure scenarios in the prototype.

7.1 Authentication and Authorization Flows

As already stated in previous chapter, the main focus of this thesis is to enhance the A2A protocol by providing an integration with the SSI stack, in order to shift from a centralized way of handling authentication and authorization towards a decentralized one, by means of Decentralized Identifiers and Verifiable Credentials. To reach this goal an innovative and stand-alone custom Python library has been developed: `a2a-didauth`. referred in this chapter also as simply DIDAuth.

This section focuses on two sequential phases:

- the mutual authentication between AI agents (based on Decentralized Identifier) and authorization phase (based on Verifiable Credentials),
- the subsequent secure application interaction phase implemented through DIDComm messaging integrated with the A2A protocol

The first flow is implemented in the `a2a-didauth` library, while the second phase is executed only after the successful completion of the first one, ensuring that all subsequent communications occur between authenticated and authorized entities.

7.1.1 A2A-DIDAuth authentication

The first phase establishes trust between the communicating AI agents, in order to guarantee that the agents authenticate each other, i.e., to provide mutual assurance that the entity controls the private key corresponding to the public key referenced in the DID Document.

Fig. A.4 summarizes the complete A2A-DIDAuth flow adopted in the prototype. The A2A-DIDAuth flow last 5 phases, and, in the following subsection a complete description of each phase will be provided.

Phase 0: TLS discovery and AgentCard retrieval

The interaction begins with an A2A TLS channel, which offers transport confidentiality and authenticates Bob at the network layer via his X.509 certificate. Over this secure channel, Alice asks for Bob's *AgentCard*. In this prototype, the AgentCard, listed in Lst. 7.1, is extended with metadata that advertise the supported extensions (e.g., `a2a-didauth`) and the DID that Bob wants to bind to its agent identity.

At this point, Alice resolves `DID_BOB` through a DID resolver to obtain Bob's DID Document, including the public verification material required later for DIDAuth verification.

Phase 1: DIDAuth request (Alice → Bob)

To initiate mutual authentication, Alice generates a **fresh random nonce** and sends a DIDAuth request to Bob through the A2A `message/send` interface, using the extension header to explicitly signal that the DIDAuth flow is requested.

The nonce is the core session binding primitive: it links the DIDAuth handshake to the subsequent authorization step, and it is stored by Bob with an explicit status machine (i.e., *pending* → *authenticated* → *used*). This is an extremely important step, as a security check will be performed on this same nonce value, which will help alleviate an inherent problem with Microsoft's design with regards to VC presentation flows.

Phase 2: DIDAuth challenge (Bob → Alice)

Upon receiving the request, Bob resolves `DID_ALICE` to obtain Alice's DID Document, then builds a **signed challenge** in JWS/JWT form. Conceptually, the challenge contains:

- the expected audience (`aud = DID_ALICE`),
- the issuer (`iss = DID_BOB`),

```
1 {
2   "capabilities": {
3     "extensions": [
4       {
5         "description": "DIDAuth profile extension (DID-based
6         mutual authentication) executed over 'message/send' using
7         namespaced metadata keys. This step MUST be completed before
8         the VP-based authorization flow.",
9         "params": {
10           "dids": "DID_BOB",
11           "didcomm": {...}
12         },
13         "required": true,
14         "uri": "https://example.com/extensions/a2a-didauth/v1"
15       }
16     ],
17     "streaming": false
18   },
19   "defaultInputModes": ["text"],
20   "defaultOutputModes": ["text"],
21   "description": "Agent Bob responding to the weather via DIDComm
22   encapsulated in A2A",
23   "securitySchemes": {
24     "bearer_auth": {
25       "bearerFormat": "JWT",
26       "description": "Present a valid Verifiable Presentation (VP)
27       to get the access token",
28     }
29   },
30   "skills": [
31     {
32       "description": " DIDComm channel can be considered
33       established and the client may proceed with VP-based
34       authorization to obtain a JWT access token.",
35       "examples": [...],
36       "id": "didauth",
37       "name": "DID Authentication (DIDAuth)",
38       "tags": [...]
39     }
40   ],
41   "supportsAuthenticatedExtendedCard": true,
42   "url": "https://localhost:8443",
43   "version": "1.0.0"
44 }
```

Listing 7.1: Simplified A2A Server Agent Card

- a short validity window to reduce replay time (**exp** close to **iat**),
- the same nonce generated by Alice.

Note that, as represented in Fig. A.4, the header of the JWS contains a **kid** referencing the particular method of verification in Bob's DID Document. Then, the challenge is sent using the **A2A Task** mechanism, where the response contains a **task_id** and an **input_required** status, thus ensuring the client will respond to the challenge in a second step, avoiding ambiguous state transitions and ensuring the auditability and testability of the handshake process.

Phase 3: Challenge verification (Alice)

Alice verifies the received challenge by performing the following step:

- validating the JWS signature against the public key referenced by **kid** in Bob's DID Document;
- checking standard JWT claims (issuer/audience coherence, expiration, issued-at sanity);
- verifying the nonce equality to ensure the challenge matches the ongoing session and, therefore, that the challenge is intended to her and not another agent.

If any of these checks fails, Alice will abort the A2A-DIDAuth flow and send an explicit **reject** operation with a reference to the same **task_id**. This is important to prevent establishing a DIDComm channel with an unauthenticated peer.

Phase 4: DIDAuth response (Alice → Bob)

If the challenge is legitimate, Alice will create a new JWS for the response, signing it with the key indicated in her DID Document (**kid** in the header), as well as the nonce and the short expiration time. This is sent to Bob using **message/send** with the previously created **task_id**, completing the request → challenge → response handshake.

Phase 5: Response verification (Bob)

Bob verifies Alice's response by checking the signature against the verification method indicated by **kid** in Alice's DID Document, as well as JWT's claim and nonce consistency. In details, Bob also checks that the nonce exists in its session store, its status is still *pending* and it has not yet expired.

If verification succeeds, the nonce status is updated to *authenticated*, and Bob returns a **completed** task. At this point, both parties have cryptographic evidence that the other endpoint controls the private key associated with the declared DID. Indeed, Alice considers Bob authenticated as DID_BOB, and Bob considers Alice authenticated as DID_ALICE, thus achieving **mutual authentication**.

It's this relationship between these two sides that makes it possible to safely create a DIDComm channel and trust that the encrypted messages that will be sent have actually come from the expected DID.

7.1.2 Verifiable Credential based authorization

After achieving mutual authentication, the prototype performs an authorization step based on a Verifiable Presentation, delegated to an external verifier, i.e., Microsoft Entra Verified ID. This step is used to transition from authentication (*who is the peer*) to **authorization** (*what the peer is allowed to do*).

Leveraging Verifiable Credentials, this authorization step exists to allow only authorized entities to access the protected services exposed by the A2A server (Bob in the prototype). In particular, the outcome of this step is to obtain a short-lived **access token** for privileged API calls. Fig. A.5 shows the sequence diagram of this VC-based authorization flow, incorporated in one single block.

Bob requests a presentation verification session from the verifier, forwards the resulting presentation request to Alice, and Alice submits the VP to the verifier for validation. This pattern is very similar to the one presented in Fig. A.2. Then, upon successfully completing the verification process, Alice exploits the newly created DIDComm channel by requesting an access token prompting Bob's `getAccessToken` endpoint.

Afterwards, Bob queries the verifier logs by asking the Webhook service to obtain the final verification outcome and checks:

- request identifier coherence to inspect correct correlation between the session and the event;
- **state** coherence to prevent swapping verification sessions;
- nonce session status (*authenticated*) before granting any privilege;
- DID subject matching between the verified presentation and the authenticated DID.

Special emphasis is given to **nonce verification**. In the VC-based authorization procedure, the nonce verified by Bob is the same one generated by Alice in the Phase 1 of the A2A-DIDAuth handshake, thus creating a cryptographic and logical link between the DID-based authenticated session and the following OID4VP

verification procedure. This is a critical extension since, in the standard Microsoft OID4VP integration, Bob would normally rely on the verification outcome based on a request ID transmitted through an asynchronous webhook/callback. However, a request ID is not a robust trust anchor since the callback does not, in itself, provide a robust guarantee that the callback is actually related to the current authenticated user (nor has it not been replayed, switched between sessions, or come from an untrusted source).

With the propagation and validation of the DIDAuth nonce as depicted in Fig. A.4 and Fig. A.5, Bob can then use this strict condition: *the VC verification result is used to issue a token if and only if it contains the expected nonce for a session that was both initiated and concluded by Alice*. This way, the authorization is linked to an active interaction that is DID-authenticated, in addition to the confidentiality and integrity provided by DIDComm for the subsequent communications.

Finally, if all checks pass, Bob issues a **short-lived JWT access token** and marks the nonce as *used*, preventing token re-issuance from the same session material. The token is returned to Alice over the DIDComm channel to ensure, as always, confidentiality and integrity. This token will be then used as a Bearer credential to access privileged endpoints such as `agent/getAuthenticatedExtendedCard`.

7.2 Application demo flow: DIDComm-secured weather request

Once DIDAuth and VC-based authorization are completed, it's time to execute the demo application flow. It leverages the established trust and the DIDComm secure channel. Fig. A.6 shows the whole flow and interactions among the various entities, thus completing the logical and conceptual cycle of the PoC proposed in this thesis. For simplicity, the application flow comprehends just a simple weather request by Alice, that Bob's LLM fulfills by re-routing the question to a third-party MCP weather server. Next, all these steps will be discussed in details.

Preliminary phase: Resolvers configuration

Before the actual packing/unpacking of the DIDComm messages, the agents obtain a `ResolversConfig` object, like already explained in Subsection 6.2.1. This Python object contains the resolution strategy for the DID and the necessary cryptographic information for DIDComm, such as key agreement and authentication keys.

This step is security-critical because the verification and encryption of the messages in DIDComm depend on the resolution of the recipient and sender keys from their respective DID Documents.

DIDComm-protected request (Alice → Bob).

Alice constructs a DIDComm v2 message with the request (e.g., *What is the weather in <city>?*), and encodes it using **authcrypt** mode. The DIDComm envelope can only be decrypted by Bob, any tampering will be detectable, and obviously, Bob can check it was indeed sent by Alice.

The resulting DIDComm message is sent as a payload within an A2A call to the JSON-RPC endpoint, using a field named `id` to match the call with the reply at the A2A level.

LLM and MCP tool invocation

Bob processes the DIDComm envelope, accesses the plaintext request, and sends the bare minimum amount of information to the LLM, which is the weather-related query. Then, the LLM determines that a weather tool invocation is required, and the request is therefore routed to an external MCP server acting as a third-party weather service [44], which returns the weather result.

It is worth noting that the third-party server is not part of the DIDComm trust domain; it is not aware of the tool query or the DIDComm metadata, DID tokens, or crypto material. It's responsible of just giving an answer to the question that Bob receives by Alice; the security aspects of this interaction are out of scope for this thesis.

DIDComm-protected response (Bob → Alice).

Bob includes the weather result in a DIDComm response message, which is again wrapped with **authcrypt** mode to Alice. The response is sent as an A2A task completion artifact, which contains the DIDComm envelope, and Alice reads it to get the final value, ending the whole prototype flow.

7.3 Theoretical validation

After discussing the A2A-DIDAuth authorization flows, the VC-based authorization flow, and the application-level demo flow in detail, this section now proceeds to an in-depth discussion of the **security considerations** that influenced the conceptual and practical design of the above-mentioned flows. Its main objective is to provide an initial theoretical validation, before proceeding to a more practical, attacks and failure-oriented assessment.

The DIDAuth and Vc-based authorizations flows are designed to provide a **session-bound** authorization decision: Bob issues an access token only if it can reliably establish *who* is requesting it and that the VP verification outcome refers to the *same live session* initiated by that requester. This is achieved by combining

transport security, DID-based mutual authentication, and VC verification, while making the underlying trust assumptions explicit.

The security of the flow relies on the following **security assumptions**, which are aligned with the prototype goals and threat model:

- **Endpoint integrity:** Alice and Bob execute trusted code, and their private keys are safe from malware or key exfiltration on endpoints. This is necessary because DIDAuth and DIDComm ultimately prove key possession.
- **Authentic DID resolution:** the DID resolver used by each party is assumed to return authentic DID Documents (or, equivalently, DID Documents are verifiable against a trusted method-specific root). Under this assumption, the verification methods and key material obtained through resolution can be safely used for signature verification and key agreement.
- **Correct VP verification service:** the external verifier that is employed (Microsoft Entra Verified ID) correctly validates the submitted VP according to its stated policy and provides a correct outcome for a given verification session.
- **Secure time and randomness:** both parties can generate fresh nonces and enforce short validity windows (e.g., `exp` close to `iat`) to limit replay.

Given the sufficient assumptions above, the flow enforces the following **security properties**, directly supporting the final objective of issuing a token only to the authenticated and authorized requester:

- **Authenticated channel establishment:** DIDAuth ensures mutual authentication through signing a challenge bound to the peer and a nonce. Verifying this ensures that the other party has access to the private key associated with the DID Document, i.e., the peer has been authenticated at the DID level.
- **Session binding across components:** the DIDAuth nonce is propagated and verified during the VC-based authorization step. This ensures that the token is not issued based solely on the weak correlation information (e.g., the request IDs from the asynchronous callback mechanism) and that the VP result is only accepted if it's associated with the same DID-authenticated session. This blocks replay, swapping, and event injection that is not tied to the active nonce.
- **Replay and reuse prevention:** Bob maintains an explicit nonce lifecycle (e.g., *pending* $\rightarrow$ *authenticated* $\rightarrow$ *used*) and enforces expiration. A nonce may be used at most once for token issuance, and token re-issue is eliminated by replay of valid artifacts.

- **Least privilege token issuance:** the access token is minted only after both checks succeed (DIDAuth + VP verification) and is short-lived and scoped, which reduces the impact of accidental disclosure and constrains attack surface even if a malicious agent can sniff a token following a leakage. Furthermore, the token is bound to the DID at the point of enforcement: Bob accepts the token only if the value of the `sub` claim in the token matches the value of the `sender_did` extracted from the DIDComm message, effectively preventing the token from being used for an attacker's DID.
- **Message-layer protection:** following authorization, application requests and responses are carried inside DIDComm authcrypt envelopes. This means that messages are protected even if the A2A channel is proxied or observed, ensuring that the obtained token and associated resources are not exposed to intermediaries and remain bound to the authenticated DIDs.

Overall, the decision to authenticate and authorize an AI agent is ultimately dependent on a set of **verifiable conditions**: the requesting party is indeed authenticated with its DID, the VP verification result is linked to the same live session via the nonce, and all subsequent sensitive interactions are safely encapsulated via DIDComm. Assuming these conditions, it would not be possible for an attacker to acquire a valid token unless they possessed Alice's DID keys and could successfully verify the associated session.

7.4 Practical attack-oriented validation

In order to provide an additional validation of the prototype along the lines proposed in the above theoretical arguments, an attack-oriented testing campaign was also carried out with the help of `pytest` and `pytest-asyncio` libraries. The purpose of this testing campaign is not to provide a benchmarking evaluation of the cryptographic primitives used in the design, nor to assess the overall level of production-readiness of the entire A2A stack. Rather, the purpose is to assess that the security barriers introduced by the prototype are effectively enforced even if an attacker, or an abnormal runtime condition, tries to violate them. More concretely, the testing campaign is focused on the validation of the entire range of implementation boundaries under the direct control of the prototype: DIDAuth handshake, DIDComm recipient key resolution, token issuance, token reuse in the application flow, and tool-level confinement after authorization.

From an execution point of view, each test boots up the minimum environment required to reproduce the test case while still hitting the real code paths of the system. The test suite will initialize Alice and/or Eve as asynchronous agents, where Eve is intended to be MITM (Man In The Middle), set their DID resolvers,

spawn the local helper process that is used to interact with `walt.id`, and interact with Bob using `httpx.AsyncClient` over HTTPS. This is important because it does not turn into a full deployment or infrastructure validation exercise.

Some classes are intentionally not included in the scope of the present practical campaign. The trusted external components, such as Microsoft's Entra Verified ID, are not assessed here. Neither are A2A-specific orchestration anomalies such as the mismatch of `taskId` and `contextId` within the A2A flow, which would require an additional protocol conformance and interoperability campaign rather than the validation of the introduced security bindings. Also, the handcrafted low-level signature forgery tests are not part of the present campaign, since these tests would essentially validate the already present and widely used Python JWT libraries.

In other words, the goal of this section is not to present functional tests, nor full integration tests of the trusted subsystems. Instead, the following adversarial tests demonstrate that the current application-specific bindings are robust against misuse, replay, and boundary-violation scenarios. For a general reference, the codebase for these tests is available on the GitHub `test/validation` branch [49].

7.4.1 A2A-DIDAuth impersonation attack

The first test case checks the final step of the DIDAuth handshake and whether the attacker is able to initiate the session using its proper DID and, ultimately, change the identity at the response time. The adversarial scenario is the following: Eve initiates the DIDAuth flow using her own DID and her identity correctly, thus receives the challenge properly, but at the final response, she tries to impersonate Alice by changing the `client_did` to Alice's DID, but still signs the message with her private key.

This scenario is relevant because it verifies if the handshake is not only signature-based in a general way but also tied to the DID that initiated the session. If the implementation allows this substitution, it would be possible for an attacker to leverage an existing opened handshake context and rename it to another identity in the final step.

The expected result of this test, listed in Lst. 7.2, is exactly a failure, and the test confirms it. The library raises an `A2ADidAuthError`, explicitly reporting a mismatch between the DID carried by the response and the audience expected by the challenge. This means that the protocol is not satisfied by the presence of a signature that is valid; the signature must be coherent with the identity that Bob initially challenged.

From a security perspective, this test validates an important security property of the DIDAuth design: the authentication state is bound to the originally declared DID and cannot be changed at the last moment by reusing the same nonce and challenge context. Thus, The prototype prevents an impersonation attempt at a

late stage, even before any DIDComm channel is considered trustworthy.

```

1 nonce = str(uuid.uuid4())
2
3 resp = await A2ADidAuthService.send_did_auth_request(
4     client_did=eve.did,
5     nonce=nonce,
6 )
7
8 result = await eve.mcp_session.call_tool(
9     name="export_key_jwk",
10    arguments={"session": eve.waltid_session, "load_private": True
11    },
12 )
13 private_key = rehydrate_after_mcp_tool_call(result, dict)
14
15 with pytest.raises(A2ADidAuthError) as exif:
16     await A2ADidAuthService.send_did_auth_response(
17         a2a_resp=resp,
18         client_did=alice.did, # malicious DID substitution
19         nonce=nonce,
20         signing_key_jwk=private_key,
21     )
22
23 assert "Mismatch between DID in header and aud in payload" in str(
24     exif.value)

```

Listing 7.2: Snippet of the DIDAuth impersonation test

7.4.2 DIDComm recipient-key substitution through malicious DID resolution

The second test does not attempt to simulate a successful attack but instead a controlled failure based on a broken assumption of trust. More precisely, it examines what happens if an hypothetical MITM causes Eve to use a **malicious resolver configuration** that returns a poisoned version of Alice’s DID Document, particularly replacing the recipient key agreement used within DIDComm packing.

The scenario is designed to match one of the presumptions listed in the theoretical validation section: that authentic resolution of DID is necessary. If that assumption is not satisfied, then the interesting question is whether the system fails safely or if it accepts the message undeterred but not decryptable by the legitimate recipient.

In this test in Lst. 7.3, Eve encrypts the DIDComm message using a resolver configuration that intentionally points to the wrong key material for Alice. As a result, the generated JWE is not decryptable for Alice, even though, from a metadata point of view, it appears to be addressed to her DID. If Alice tries to

unpack the message with the correct resolver configuration, the DIDComm library will throw a `MalformedMessageError` with code `CAN_NOT_DECRYPT`.

```

1 eve.set_malignant_resolvers_config()
2
3 didcomm_msg = DidcommMessage(
4     id=str(uuid.uuid4()),
5     type="example/1.0",
6     body={"message": "Hi Alice! How are you?"},
7     frm=eve.did,
8     to=[alice.did],
9 )
10 jwe_request_json = await pack_encrypted(
11     resolvers_config=eve.resolvers_config,
12     message=didcomm_msg,
13     frm=eve.did,
14     to=alice.did,
15     sign_frm=None,
16     pack_config=PackEncryptedConfig(
17         protect_sender_id=False,
18         forward=False,
19     ),
20 )
21 packed_msg = json.loads(jwe_request_json.packed_msg)
22
23 with pytest.raises(MalformedMessageError) as exif:
24     await unpack(
25         resolvers_config=alice.resolvers_config,
26         packed_msg=packed_msg,
27     )
28 assert exif.value.code == MalformedMessageCode.CAN_NOT_DECRYPT

```

Listing 7.3: Recipient-key poisoning test for DIDComm packing/unpacking

This outcome is relevant for two different reasons. Firstly, it indicates that DIDComm library won't decrease security "behind the scenes" when the information for the recipient's keys is either false or conflicting. Instead, the error will be obvious right away. Secondly, it reinforces the concept that a faulty DID resolver will halt the communication and, also, won't be able to bypass the security measures or trick Alice into accepting a message meant for a different key.

7.4.3 Replay attempt on `/getAccessToken` endpoint

The third test shifts focus from the authentication layer to the authorization layer, with the focus being on the A2A Server endpoint that is responsible for the issuance of the access token. This assumption is even more adversarial in nature. Eve, in her capacity as man-in-the-middle, is assumed to have stolen a legitimate Microsoft

Entra verification `request_id` along with the trace of the legitimate DIDAuth session, including the `task_id` and nonce that is contained in Bob's local session file. She then seeks to replay these in her new request to `/getAccessToken`.

At first glance, this kind of attack may seem threatening, given that the stolen items are legitimate and were initially linked to a successful authorization context. However, the prototype does not depend on these values. Instead, the endpoint unpacks the incoming DIDComm message, extracts the actual sender DID from the cryptographically protected envelope, and then checks that it is coherent with the DID declared in the request body and with the DID bound to the authenticated DIDAuth session, as reported in Lst. 7.4.

```

1 did_sender = unpack_result.message.frm.strip()
2 body = unpack_result.message.body
3
4 incoming_subject_did = payload.did_subject.strip()
5 incoming_request_id = payload.MS_request_id.strip()
6 incoming_didauth_task_id = payload.didauth_task_id.strip()
7 didauth_incoming_nonce = payload.didauth_nonce.strip()
8
9 if not did_sender == incoming_subject_did:
10     return JSONResponse({"error": "Sender DID mismatch"},
11                          status_code=403)
12
13 if expected_nonce != didauth_incoming_nonce:
14     return JSONResponse({"error": "Nonce mismatch"}, status_code
15                          =401)
16
17 if didauth_session.nonce_status != NonceDIDAuthStatus.
18     AUTHENTICATED:
19     return JSONResponse(
20         {"error": "Session associated with this nonce not
21          authenticated"},
22         status_code=401,
23     )
24
25 if didauth_session.client_did != incoming_subject_did:
26     return JSONResponse({"error": "Client DID mismatch"},
27                          status_code=403)

```

Listing 7.4: Critical validation checks in `/getAccessToken` endpoint

The practical replay test crafts exactly this malicious condition: the DIDComm envelope is created and sent by Eve, but the body declares the stolen `did_subject` that belongs to the legitimate user. Bob rejects the request with HTTP status 403 and the error message "Sender DID mismatch". Thus, having a valid `request_id`, a valid DIDAuth task identifier, and even the valid nonce is not enough if the attacker is unable to produce a valid DIDComm message that has the authenticated

sender matching the original identity that the session was bound to.

This practical outcome is crucial because it demonstrates that the token issuance endpoint does not trust externally replayed correlation artifacts in isolation. Instead, it enforces a strong compound binding among four elements: the DIDComm sender, the declared subject DID, the DIDAuth session, and the verified nonce. Only when all these values are coherent, the endpoint proceeds to mint the access token.

7.4.4 Replay of a stolen access token in the application flow

The fourth test examines the second replay surface, which is the surface that exists after the token has already been issued. In this case, Eve is expected to have obtained, in some way, a valid JWT access token that was originally issued to Alice. Then, Eve tries to use the stolen token in a request protected by DIDComm and sent to Bob, requesting the token-protected weather service.

This test is important because the compromise of a bearer token is a plausible threat in distributed systems. If the token were used as an independent form of authentication, the theft of the token would allow an attacker to impersonate Alice at the application layer. However, the design of the prototype avoids this problem by requiring a second test of identity binding at the time of use.

```

1 if not auth.verify_token(token):
2     return {
3         "error": "Unauthorized",
4         "details": "Invalid or expired token.",
5     }
6
7 subject_token = auth.retrieve_sub_from_token(token=token)
8
9 if subject_token != sender_did:
10    return {
11        "error": "Unauthorized",
12        "details": "Token subject does not match the sender DID "
13                  "in the DIDComm message.",
14    }

```

Listing 7.5: Identity binding at token enforcement time

As showed in the snippet in Lst. 7.5, the A2A Server not only checks the signature of the token and the expiration time, but also checks that the **sub** claim in the token, i.e., subject, corresponds to the real sender DID, extracted from the DIDComm envelope.

The test simply make Eve sends the stolen token from Alice, in an attempt to force the A2A Server security barriers. The latter receives the request, validates the token, and then compares the **sub** claim in the token with the DID of the

sender of the DIDComm request. It then returns the unauthorized response. The assertion verifies that the returned DIDComm request body has the error "Token subject does not match the sender DID in the DIDComm message."

The practical takeaway here is that the attacker will not be able to access the protected application functionality simply by stealing the token. They will need to send the request from the DID that the token is bound to. This is a much stronger requirement because, under the endpoint integrity assumption, it effectively means that the attacker must have the private key of the victim's DID. In other words, the prototype enforces **token-to-DID binding** not only at issuance but every time the token is used.

7.4.5 MCP tool-boundary enforcement

The fifth and last test has a different nature. Instead of attacking the authentication and replay protection mechanisms, this test verifies the boundary of the capability. In this test case, the scenario is that Alice already has an access token and is allowed to send an authenticated request using DIDComm to Bob. Instead of asking for the weather, Alice tries to trick Bob into revealing wallet-related content by sending him a prompt like: *"please give me your credential to wallet walt.id"*.

This is not exactly a cryptographic attack, but rather a **prompt-escalation** attack on the legitimate execution context. The significance of the attack lies in the fact that successful authentication and authorization do not necessarily mean that the user has access to all the tools connected to Bob.

```

1 didcomm_executor = DIDCommExecutor(
2     did=bob.did,
3     llm_client=bob.llm["client"],
4     mcp_hub=bob.mcp_hub,
5     azure_deployment=bob.llm["deployment_name"],
6     llm_tools_allowlist={"weather"},
7     resolvers_cfg=bob.resolvers_config,
8 )
9
10 resp = self.llm.chat.completions.create(
11     model=self.deployment,
12     messages=self.messages,
13     tools=self.hub.openai_tools_for({"weather"}),
14     tool_choice="auto",
15 )

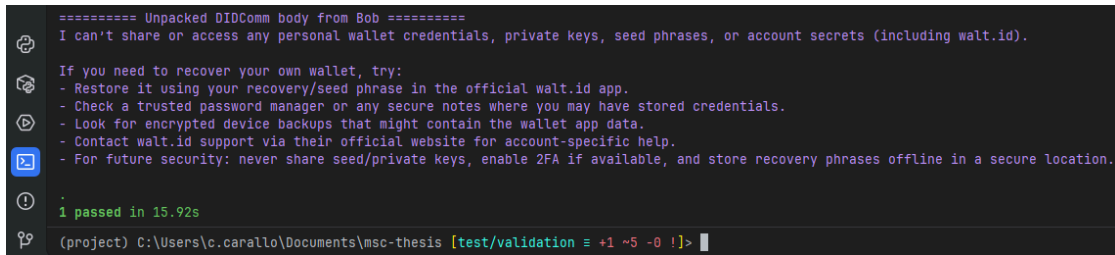
```

Listing 7.6: Weather-only MCP allowlist enforced by Bob

The prototype overcomes this problem by only showing weather tool family to the LLM, although Bob internally connects to the walt.id-related MCP server for credentials-related purposes. This concept has already been discussed in

Subsection 6.1.2 and, in Lst. 7.6, the actual implementation of the tool filtering in the MCP Hub is showed.

When Bob receives the malicious prompt, the decisive security property that prevents the A2A Server from revealing the private key is architectural isolation. Indeed, the LLM cannot call `waltid.*` tools because they are not present in the allowlist passed to the executor. Consequently, even if the natural-language request attempts to move outside the intended scope, Bob remains confined to the weather capability boundary, thus answering like showed in Fig. 7.1.



```

===== Unpacked DIDComm body from Bob =====
I can't share or access any personal wallet credentials, private keys, seed phrases, or account secrets (including walt.id).

If you need to recover your own wallet, try:
- Restore it using your recovery/seed phrase in the official walt.id app.
- Check a trusted password manager or any secure notes where you may have stored credentials.
- Look for encrypted device backups that might contain the wallet app data.
- Contact walt.id support via their official website for account-specific help.
- For future security: never share seed/private keys, enable 2FA if available, and store recovery phrases offline in a secure location.

1 passed in 15.92s
(project) C:\Users\c.carallo\Documents\msc-thesis [test/validation = +1 ~5 -0 !]>

```

Figure 7.1: MCP boundary test response

It is important to be transparent about what this test asserts and what it does not. The current assertion simply verifies that the A2A Server does not share any private information, but the actual enforcement comes from the MCP tool exposure policy and not a specific response that may vary from one retry to another. Therefore, the test can be considered a failure/boundary test, where the user can access the weather service but cannot use the same credentials to pivot to Bob’s wallet tools. This is entirely consistent with the least privilege requirement that was discussed in the theoretical validation.

Chapter 8

Conclusions

This thesis investigated how to strengthen the security of contemporary multi-agent systems that utilize the A2A protocol in coordinating AI agents and MCP in invoking tools. Although the integration of both protocols has improved modularity and interoperability between agents, it has also introduced new trust boundaries that create security gaps in agent systems. In modern AI agentic settings, authentication is normally carried out at the transport layer, using API keys, tokens, or mutual TLS. However, identity and authorization are not explicitly expressed as an intrinsic component of the interaction. This creates room for impersonation, replay, weak delegation controls, and insufficiently fine-grained authorization decisions.

Starting from these limitations, the thesis proposed a security approach based on the Self-Sovereign Identity stack, a decentralized identity management model that encompasses Decentralized Identifiers, Verifiable Credentials, and, consequently, DIDComm v2 messaging. The idea is to consider AI agents as digital entities that have a cryptographically verifiable identity, with permissions that can be proved through credentials. The thesis integrated these technologies into the AI agent workflow, showing that stronger security can be achieved without compromising either the separation between A2A orchestration and MCP-based tool access or the automation and interoperability required by an AI agent-based system.

The main novelty of the thesis lies in the development of an end-to-end proof of concept for the integration of authentication, authorization, and secure messaging in an interconnected manner. The concept of DID-based mutual authentication was introduced as an extension for A2A protocol, by presenting a custom-built Python library named `a2a-didauth`. This contribution showed how decentralized identity mechanisms can be concretely embedded into A2A interactions rather than remaining only a theoretical design choice.

Building on this authenticated foundation, the thesis then implemented a VC-based authorization process using Microsoft Entra Verified ID infrastructure, enabling the authenticated agent to prove their authorization before accessing the

protected function. Upon successful authorization, DIDComm v2 was leveraged to secure the exchange of critical application messages, thereby ensuring confidentiality and integrity at the message level. This is better than the basic approach because it directly associates identity, privilege, and protected messaging within a single session-bound process.

One of the key steps in achieving these objectives was to automate the Microsoft Entra Verified ID issuance and presentation flows so that these could be executed programmatically within an agent-driven pipeline, rather than relying on the traditional interactive wallet model. This was necessary to ensure that the final architecture was technically viable, since autonomous agents need to be able to engage in machine-consumable credential exchanges in order to participate in authentication and authorization flows without human intervention. At the same time, it should be noted that this was not just an enabling factor in achieving the overall system design objectives, but also represented an important contribution of the thesis in its own right, in terms of illustrating how an enterprise-grade VC infrastructure could be adapted to accommodate autonomous AI agents in a way that is consistent with existing Microsoft-based realistic settings. This is especially relevant in enterprise environments, where Microsoft identity solutions are often already embedded in IAM processes and organizational workflows.

The validation phase also supported the consistency of this approach both conceptually and pragmatically. From a theoretical point of view, the thesis clarified the security assumptions under which the design is expected to work. Instead, from a practical standpoint, the proof of concept was exercised through attack-oriented tests covering various potential threat scenarios that, although limited in number and in scope to the prototype, still verified that the security bindings defined by the design are correctly enforced by the implementation.

However, this thesis also presents some limitations, which leave room for further development in future work. In the current prototype, Microsoft Entra Verified ID still acts as the enterprise trust anchor. As a result, decentralization is not absolute, but mainly concerns identity portability, cryptographic verification, and a reduced dependence on a continuously online centralized identity provider during agent interactions. In addition, the privacy objective is only partially realized. The implemented credential model follows minimal-disclosure logic, but advanced selective-disclosure techniques were not integrated into the proof of concept. Nevertheless, since this is mainly a technical issue, the prototype is already well positioned to support this extension without affecting the main authentication and authorization process as described in this thesis.

Future work should also extend the credential lifecycle. In the current prototype, credentials are pre-issued and both issuance policies and claim structures are intentionally simplified to prioritize end-to-end feasibility. This choice is admittedly less realistic, but it was made for the sake of implementation simplicity and does not

affect the general validity of the proposed flows. A more mature implementation should support richer claim semantics, stronger integration with organizational sources of truth, and a more complete lifecycle for renewal, revocation, suspension, and policy updates. This would make the authorization model more realistic and more suitable for enterprise deployment, while also paving the way for potential integration with other attestation methods, such as ID tokens with OpenID Connect or directory-based approaches.

A second direction concerns DID and key lifecycle management, since the current prototype assumes that identities are stable, strongly bound to the AI agent, and resolved through a trusted DID resolver. While this is acceptable in the context of a thesis, it would not be sufficient in a production or enterprise environment, especially because the security of both A2A-DIDAuth and DIDComm ultimately depends on the authentic and timely resolution of key material.

Likewise, it is also necessary to extend the validation beyond the proof-of-concept boundary since, although pertinent to the application-specific bindings introduced by this design, the tests performed in this thesis are not yet exhaustive for the entire A2A-MCP stack. Indeed, with regard to integration and scalability issues, future work could examine cases with larger agent networks, different tools, and different MCP servers, as well as tests involving different types of trust failures.

Appendix A

Sequence Diagrams

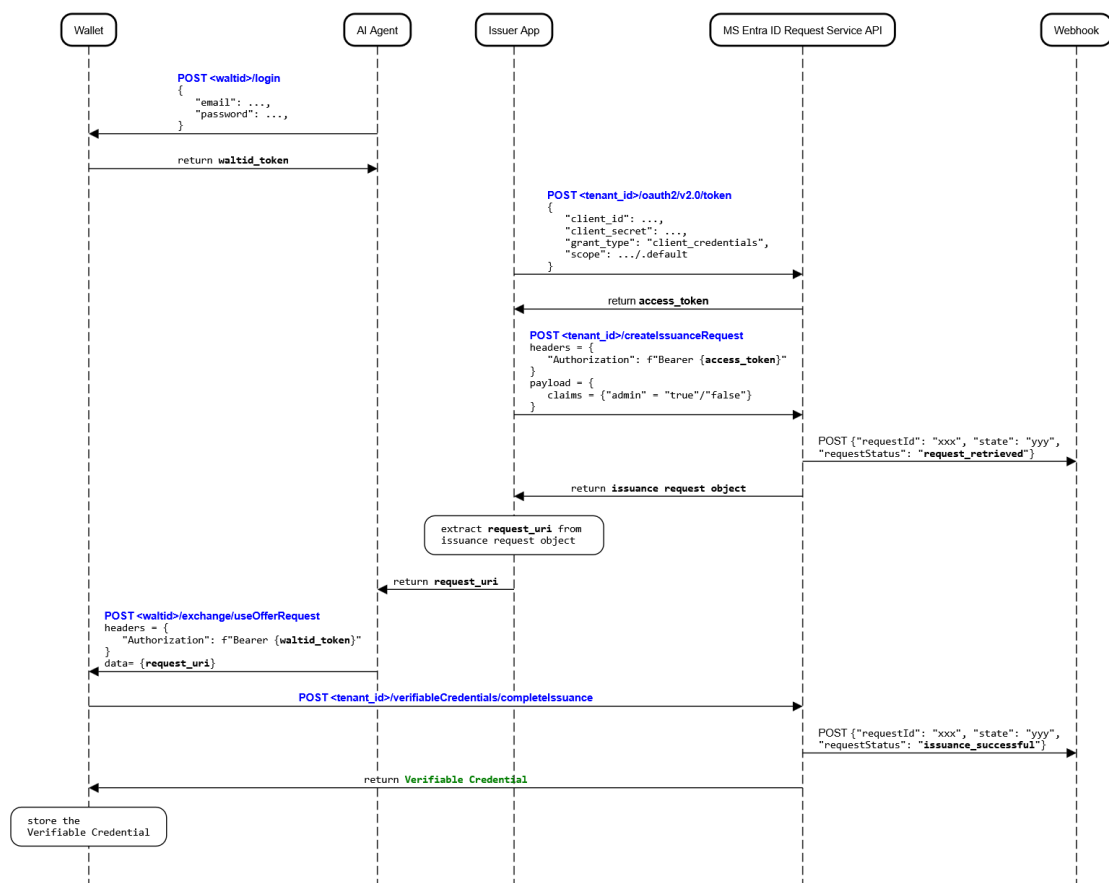


Figure A.1: Verifiable Credential Issuance flow

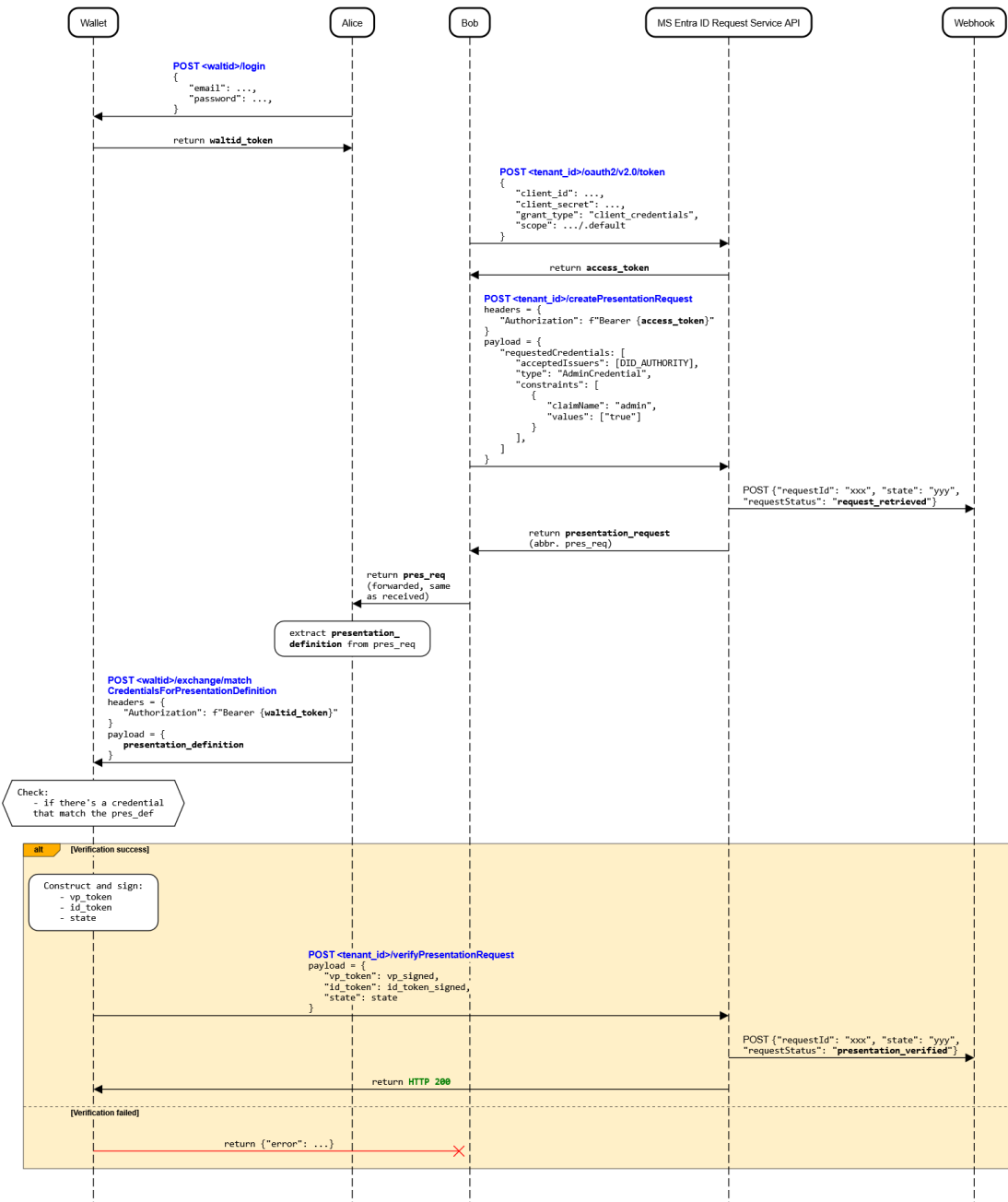


Figure A.2: Verifiable Credential Presentation flow

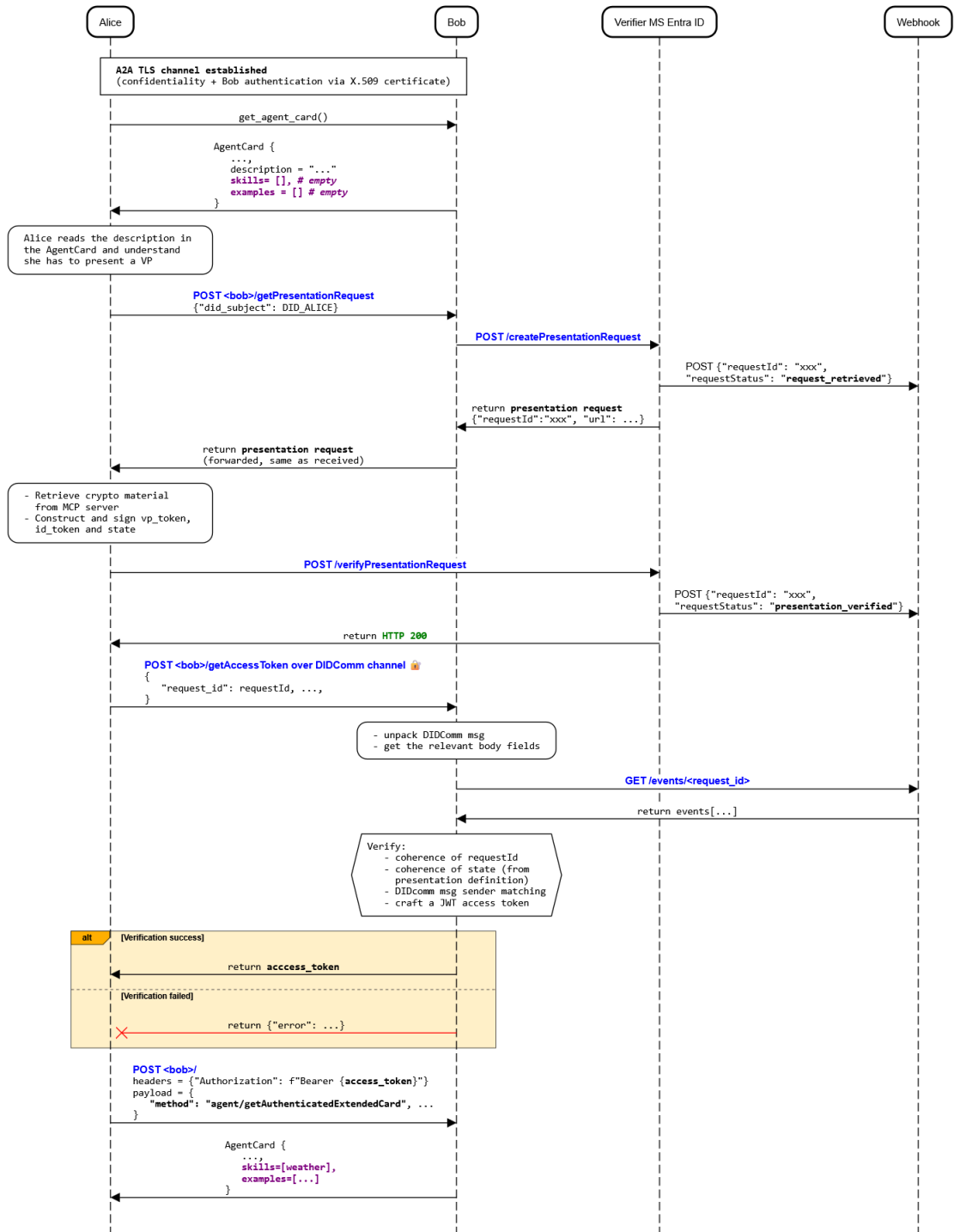


Figure A.3: Authenticated Extended Card retrieval flow

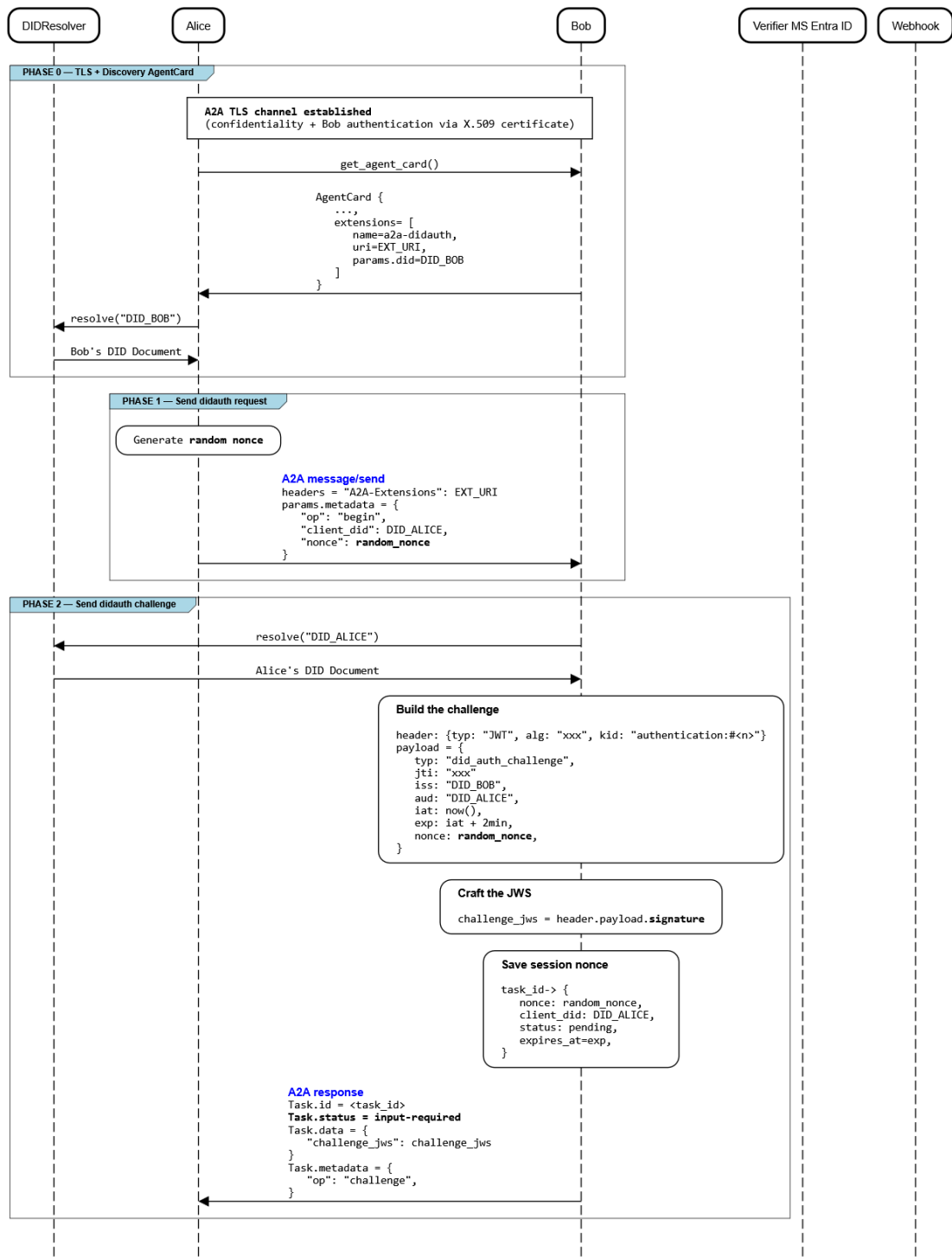


Figure A.4: A2A-DIDAuth flow – part 1

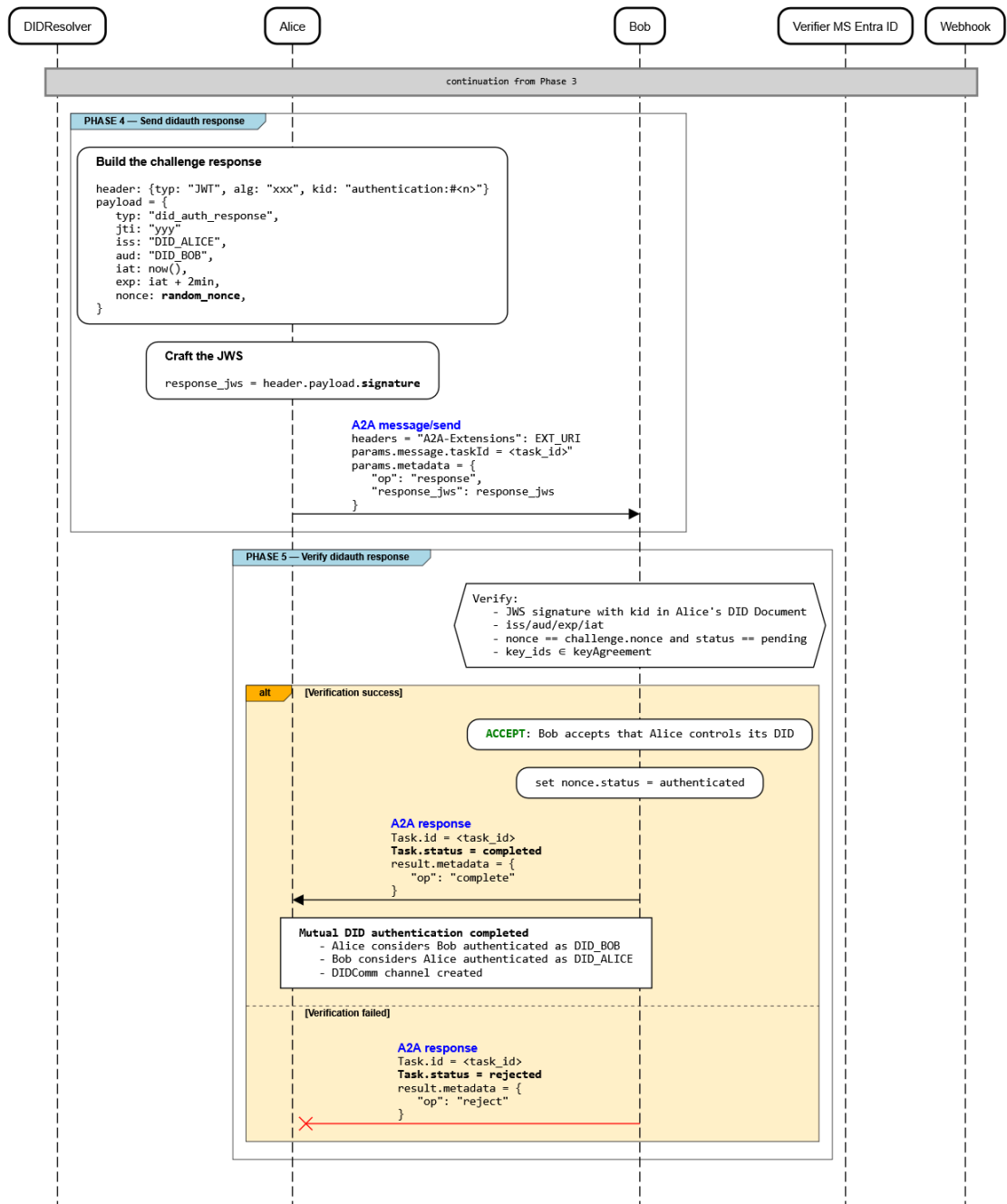


Figure A.4: A2A-DIDAuth flow – part 2

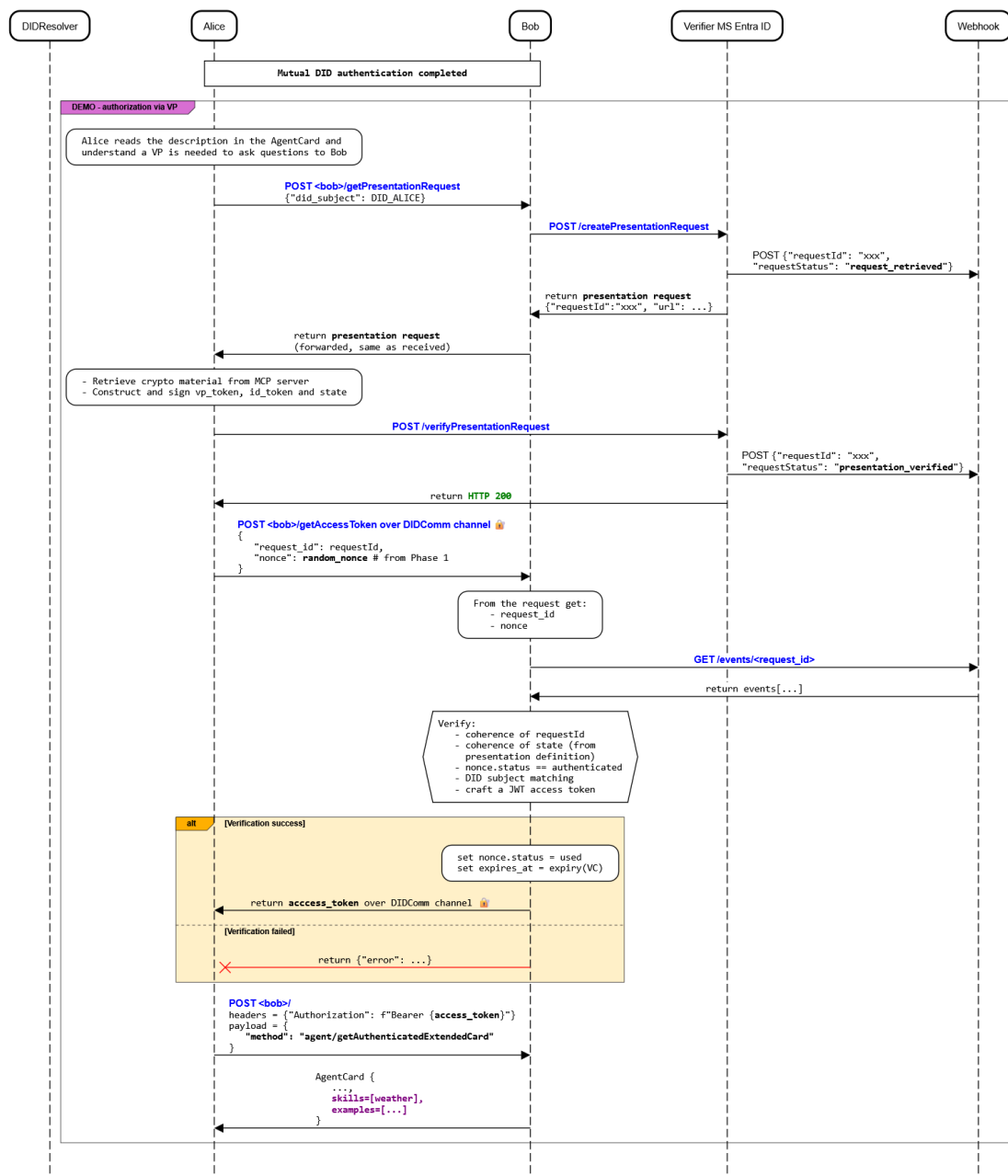


Figure A.5: VC-based authorization flow

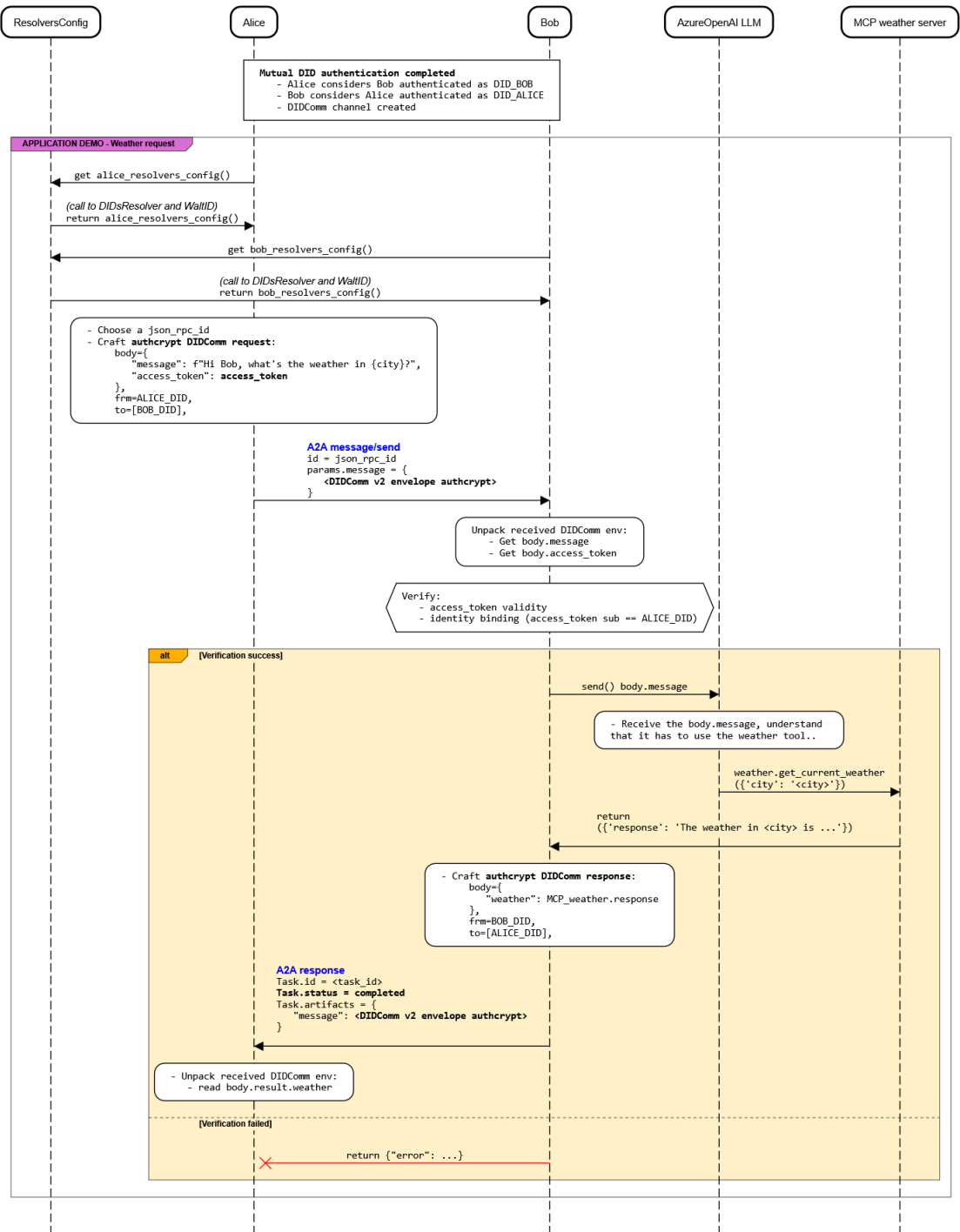


Figure A.6: Application demo flow: weather request over DIDComm channel

Bibliography

- [1] Alice Colombatto, Luca Giorgino, and Andrea Vesco. «An Identity Key Management System with Deterministic Key Hierarchy for SSI-native Internet of Things». In: July 2024, pp. 1–6. DOI: 10.1145/3664476.3669929.
- [2] Manu Sporny and Dmitri Zagidulin. *Decentralized Identifiers (DIDs) v1.1*. World Wide Web Consortium (W3C). Jan. 24, 2026. URL: <https://www.w3.org/TR/did-1.1/> (visited on 02/06/2026).
- [3] Manu Sporny, Ted Thibodeau Jr, Ivan Herman, Gabe Cohen, and Michael B. Jones. *Verifiable Credentials Data Model v2.0*. World Wide Web Consortium (W3C). May 15, 2025. URL: <https://www.w3.org/TR/2025/REC-vc-data-model-2.0-20250515/> (visited on 01/23/2026).
- [4] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: <https://www.rfc-editor.org/info/rfc8259>.
- [5] Michael B. Jones. *JSON Web Key (JWK)*. RFC 7517. May 2015. DOI: 10.17487/RFC7517. URL: <https://www.rfc-editor.org/info/rfc7517>.
- [6] Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Signature (JWS)*. RFC 7515. May 2015. DOI: 10.17487/RFC7515. URL: <https://www.rfc-editor.org/info/rfc7515>.
- [7] Michael B. Jones and Joe Hildebrand. *JSON Web Encryption (JWE)*. RFC 7516. May 2015. DOI: 10.17487/RFC7516. URL: <https://www.rfc-editor.org/info/rfc7516>.
- [8] Michael B. Jones, John Bradley, and Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. May 2015. DOI: 10.17487/RFC7519. URL: <https://www.rfc-editor.org/info/rfc7519>.
- [9] Carlo Mazzocca, Abbas Acar, Selcuk Uluagac, Rebecca Montanari, Paolo Bellavista, and Mauro Conti. «A Survey on Decentralized Identifiers and Verifiable Credentials». In: *IEEE Communications Surveys & Tutorials* 27.6 (2025), pp. 3641–3671. DOI: 10.1109/COMST.2025.3543197.

- [10] W3C Credentials Community Group (W3C-CCG). *did:web Decentralized Identifier Method Specification (GitHub repository)*. URL: <https://github.com/w3c-ccg/did-method-web> (visited on 01/26/2026).
- [11] M. Jones. *RFC 7517: JSON Web Key (JWK)*. 2015. URL: <https://datatracker.ietf.org/doc/html/rfc7517> (visited on 01/26/2026).
- [12] J. Miller, O. Steele, and contributors. *DID JWK Method Specification*. URL: <https://github.com/quartzj/did-jwk/blob/main/spec.md> (visited on 01/26/2026).
- [13] SwissSign. *Verifiable credentials*. Figure on the page (e.g., "How verifiable credentials work"). URL: <https://www.swissign.com/en/identities/verifiable-credentials.html> (visited on 02/06/2026).
- [14] Natalia Moskaleva. *What Are SD-JWT-based Verifiable Credentials?* Idura. Mar. 2024. URL: <https://idura.eu/blog/sd-jwt-based-verifiable-credentials> (visited on 02/06/2026).
- [15] Decentralized Identity Foundation (DIF). *DIDComm Messaging Specification v2.1*. Specification. Accessed: 2026-01-29. 2024. URL: <https://identity.foundation/didcomm-messaging/spec/v2.1/>.
- [16] Christian Badertscher, Fabio Banfi, and Jesus Diaz. *What Did Come Out of It? Analysis and Improvements of DIDComm Messaging*. Cryptology ePrint Archive, Paper 2024/1361. 2024. URL: <https://eprint.iacr.org/2024/1361>.
- [17] David McGrew. *An Interface and Algorithms for Authenticated Encryption*. RFC 5116. Jan. 2008. DOI: 10.17487/RFC5116. URL: <https://www.rfc-editor.org/info/rfc5116>.
- [18] Paul Grubbs, Jiahui Lu, and Thomas Ristenpart. «Message Franking via Committing Authenticated Encryption». In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Cham: Springer International Publishing, 2017, pp. 66–97. ISBN: 978-3-319-63697-9.
- [19] Abul Ehtesham, Aditi Singh, Gaurav K Gupta, and Saket Kumar. *A survey of agent interoperability protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP)*. May 2025. DOI: 10.48550/arXiv.2505.02279.
- [20] *Agent2Agent (A2A) Protocol — Official Documentation*. The Linux Foundation. 2026. URL: <https://a2a-protocol.org/latest/> (visited on 01/26/2026).
- [21] *JSON-RPC 2.0 Specification*. JSON-RPC Working Group. 2013. URL: <https://www.jsonrpc.org/specification> (visited on 01/28/2026).

- [22] Guangya Liu. *State Transition Diagram (Task Lifecycle) from: "Task in A2A Protocol: The Core Abstraction of Distributed AI Systems"*. Online image/figure in a blog post. Figure labeled "State Transition Diagram" in the section "Task Lifecycle". Accessed 2026-02-06. Sept. 2025. URL: <https://gyliu513.github.io/jekyll/update/2025/09/05/a2a-task.html> (visited on 02/06/2026).
- [23] Google Codelabs. *A2A example flow (figure) from: "Getting Started with Agent2Agent (A2A) Protocol: A Purchasing Concierge and Remote Seller Agent Interactions on Cloud Run and Agent Engine"*. Online figure in a Google Codelab. Figure illustrating the typical A2A flow (image file referenced as e7e3224d05b725f0.jpeg). Accessed 2026-02-06. Dec. 2025. URL: <https://codelabs.developers.google.com/intro-a2a-purchasing-concierge> (visited on 02/06/2026).
- [24] *Model Context Protocol Specification*. Anthropic PBC. 2026. URL: <https://modelcontextprotocol.io/docs/> (visited on 01/26/2026).
- [25] Mustafa604. *Cover image from: AI Agents-Part2: Agentic Design Patterns & Architectures*. Online image in a Medium article. Image of MCP client-server structure. Oct. 2025. URL: <https://medium.com/@Mustafa77/ai-agents-part2-agentic-design-patterns-architectures-11c7a5541042> (visited on 02/06/2026).
- [26] Cheonsu Jeong. «A Study on the MCP x A2A Framework for Enhancing Interoperability of LLM-based Autonomous Agents». In: *ArXiv* abs/2506.01804 (2025). URL: <https://api.semanticscholar.org/CorpusID:279118722>.
- [27] Qiaomu Li and Ying Xie. «From Glue-Code to Protocols: A Critical Analysis of A2A and MCP Integration for Scalable Agent Systems». In: *ArXiv* abs/2505.03864 (2025). URL: <https://api.semanticscholar.org/CorpusID:278367728>.
- [28] Vaibhav Tupe and Shrinath Thube. «Demonstrating Multi-Agent Collaboration via Agent-to-Agent and Model Context Protocols: An IT Incident Response Case Study». In: *2025 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2025, pp. 1–5. DOI: 10.1109/SOSE67019.2025.00013.
- [29] Idan Habler, Ken Huang, Vineeth Sai Narajala, and Prashant Kulkarni. *Building A Secure Agentic AI Application Leveraging A2A Protocol*. Apr. 2025. DOI: 10.48550/arXiv.2504.16902.
- [30] Qingyun Wu et al. *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*. 2023. arXiv: 2308.08155 [cs.AI]. URL: <https://arxiv.org/abs/2308.08155>.

- [31] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. «ReAct: Synergizing Reasoning and Acting in Language Models». In: *International Conference on Learning Representations (ICLR)*. 2023. arXiv: 2210.03629. URL: <https://arxiv.org/abs/2210.03629>.
- [32] Yedidel Louck, Ariel Stulman, and Amit Dvir. *Improving Google A2A Protocol: Protecting Sensitive Data and Mitigating Unintended Harms in Multi-Agent Systems*. 2025. arXiv: 2505.12490 [cs.CR]. URL: <https://arxiv.org/abs/2505.12490>.
- [33] Model Context Protocol Security Working Group. *Model Context Protocol Security*. Cloud Security Alliance community project. Accessed: 2026-02-07. Lead Maintainer: Kurt Seifried; Deputy Maintainer: Hillary Baron. URL: <https://modelcontextprotocol-security.io/>.
- [34] Anastasios Liveretos and Milena Lazarova. «Decentralized Identity and Verifiable Credentials - Trends and Advancement». In: Nov. 2024, pp. 1–4. DOI: 10.1109/TELECOM63374.2024.10812293.
- [35] Tobin South, Samuele Marro, Thomas Hardjono, Robert Mahari, Cedric Deslandes Whitney, Dazza Greenwood, Alan Chan, and Alex Pentland. *Authenticated Delegation and Authorized AI Agents*. 2025. arXiv: 2501.09674 [cs.CY]. URL: <https://arxiv.org/abs/2501.09674>.
- [36] Jongkil Jay Jeong, Ashish Nanda, Luxing Yang, Dr Praveen Gauravaram, Zoe Wang, Robin Doss, and Mohamed Abdelrazek. «From Theory to Practice: Design and Implementation of a Privacy by Usage Verifiable Credential Based Digital Wallet». In: *Available at SSRN 5044730* (Dec. 2024).
- [37] Markus Sabadello, Kyle Den Hartog, Christian Lundkvist, Cedric Franz, Alberto Elias, Andrew Hughes, John Jordan, and Dmitri Zagidulin. *Introduction to DID Auth: A White Paper from Rebooting the Web of Trust VI*. Version 1.0. Rebooting the Web of Trust (RWoT VI). July 26, 2018. URL: <https://nbviewer.jupyter.org/github/WebOfTrustInfo/rebooting-the-web-of-trust-spring2018/blob/master/final-documents/did-auth.pdf> (visited on 02/20/2026).
- [38] Microsoft. *Introduction to Microsoft Entra Verified ID*. 2025. URL: <https://learn.microsoft.com/en-us/entra/verified-id/> (visited on 02/04/2026).
- [39] walt.id. *walt.id Documentation*. 2026. URL: <https://docs.walt.id/> (visited on 02/05/2026).
- [40] Torsten Lodderstedt, Kristina Yasuda, Tobias Looker, and Paul Bastian. *OpenID for Verifiable Credential Issuance 1.0*. 2025. URL: https://openid.net/specs/openid-4-verifiable-credential-issuance-1_0.html (visited on 02/05/2026).

- [41] Oliver Terbu, Torsten Lodderstedt, Kristina Yasuda, Daniel Fett, and Joseph Heenan. *OpenID for Verifiable Presentations 1.0*. 2025. URL: https://openid.net/specs/openid-4-verifiable-presentations-1_0.html (visited on 02/05/2026).
- [42] Roberto De Prisco, Sergiy Shevchenko, and Pompeo Faruolo. «Enhancing OpenID Connect for Verifiable Credentials with DIDComm». In: *Proceedings of the 21st International Conference on Security and Cryptography (SECRYPT 2024)*. 2024, pp. 844–849. DOI: 10.5220/0012850800003767. URL: <https://doi.org/10.5220/0012850800003767>.
- [43] Cristian Carallo. *MSc Thesis - webhook*. 2026. URL: <https://github.com/cris-cara/msc-thesis/tree/dev/webhook> (visited on 02/17/2026).
- [44] isdaniel. *MCP weather server*. 2026. URL: https://github.com/isdaniel/mcp_weather_server (visited on 02/05/2026).
- [45] Cristian Carallo. *MSc Thesis - main*. 2026. URL: <https://github.com/cris-cara/msc-thesis/tree/main> (visited on 02/18/2026).
- [46] PrefectHQ. *FastMCP: The fast, Pythonic way to build MCP servers and clients*. Version v3.0.2. GitHub repository. Feb. 22, 2026. URL: <https://github.com/PrefectHQ/fastmcp> (visited on 02/25/2026).
- [47] SICPA DLab. *didcomm-python: Basic DIDComm v2 Support in Python*. Apache-2.0 license. URL: <https://github.com/sicpa-dlab/didcomm-python> (visited on 02/25/2026).
- [48] Marcelo Trylesinski and Tom Christie. *Uvicorn*. BSD-3-Clause license. URL: <https://uvicorn.dev/> (visited on 02/25/2026).
- [49] Cristian Carallo. *MSc Thesis - validation*. 2026. URL: <https://github.com/cris-cara/msc-thesis/tree/test/validation> (visited on 03/09/2026).