

Technical University Berlin

Information Systems Engineering

Fakultät IV

Einsteinufer 17

10587 Berlin

<https://www.tu.berlin/ise>



Internet of Services: Lab

AI Agents with Digital Identities

Enis Mert Kuzu,
Alexander Gaballa,
Buse Varkan,
Dennis Enrique Gehrman

21.07.2025

Supervised by
Sandro Rodriguez Garzon & Awid Vaziry

Contents

List of Figures	iii
List of Tables	iv
1 Introduction	1
1.1 Motivation	1
1.2 Objective	1
1.3 Scope	1
2 Technical Fundamentals	3
2.1 Description of Technologies	3
2.1.1 LangGraph Implementation in Domain A	3
2.1.2 AutoGen Implementation in Domain B	8
2.1.3 Google A2A	13
2.1.4 Decentralized Identity Mechanism	16
2.1.5 Description of Signature and Verification Tools	17
2.2 Setup and how to run the Code	18
3 Evaluation	20
3.1 Test Environment	20
3.2 Findings	21
3.2.1 Unexpected LLM Behavior	21
3.2.2 Evaluation: Issues and Problems	22
3.3 Performance Measurements	24
3.3.1 Model: GPT-4.1	26
3.3.2 Model: GPT-4.1-mini	27
3.3.3 Model: GPT-4o-mini	28
3.3.4 Model: O4-mini	29
4 Conclusion	30
List of Acronyms	33
Bibliography	34

List of Figures

1.1	System Overview	2
2.1	Architecture of the AutoGen-based agents in Domain A	5
2.2	Agent X verifies Agent A using decentralized credentials	6
2.3	Agent A verifies Agent X and receives a credential	7
2.4	Overview of the MCP-based architecture in Domain B	9
2.5	Architecture of the AutoGen-based agents in Domain B	10
2.6	Agent Y verifies Agent B using decentralized credentials	11
2.7	Agent B verifies Agent Y and receives a credential	12
2.8	Agent interaction via A2A protocol	14
3.1	Comparison of Mean Completion Time per Step	25
3.2	Model Success Rate per Step	25

List of Tables

3.1	Agent Communication Step-wise Token and Time Metrics	26
3.2	Time to Completion and Success Metrics Table	26
3.3	Agent Communication Step-wise Token and Time Metrics for gpt-4.1-mini . . .	27
3.4	Time to Completion and Success Metrics Table	27
3.5	Agent Communication Step-wise Token and Time Metrics for gpt-4o-mini	28
3.6	Time to Completion and Success Metrics Table	28
3.7	Agent Communication Step-wise Token and Time Metrics for o4-mini	29
3.8	Time to Completion and Success Metrics Table	29

1 Introduction

1.1 Motivation

As artificial intelligence (AI) agents become increasingly autonomous and interconnected, their ability to operate securely and reliably across digital ecosystems becomes paramount. Whether scheduling meetings, negotiating contracts, or performing delegated tasks, these agents must interact in a trustworthy and verifiable manner—especially as they begin to represent users or organizations in high-stakes scenarios. However, the current lack of robust mechanisms for identity verification and access control in multi-agent systems leaves these interactions vulnerable to impersonation, data leaks, and unauthorized behavior.

To address these concerns, decentralized identity standards such as Decentralized Identifiers (DIDs) and Verifiable Credentials (VCs), developed by the W3C, offer a promising solution. These technologies enable AI agents to verify each other's identities and permissions autonomously, without relying on centralized authorities. In this project, we explore how combining these identity mechanisms with large language model (LLM)-based AI agents can create a new generation of intelligent, secure, and self-governing agent ecosystems.

1.2 Objective

The primary objective of this project is to design, implement, and evaluate a secure, decentralized identity management system for multi-agent interactions. Our solution enables agents powered by modern LLM frameworks—such as LangGraph and AutoGen—to identify, authenticate, and authorize each other across different security domains using DIDs and VCs. Specifically, we aim to:

- Develop agents that possess decentralized digital identities (DIDs)
- Implement intra-domain and inter-domain interactions with verifiable credential flows
- Integrate tools for issuing, presenting, and verifying credentials
- Establish secure cross-domain communication using agent-to-agent (A2A) protocols
- Evaluate different LLMs and frameworks in terms of performance and success rates

Our approach emphasizes full credential lifecycle implementation, from DID resolution and signature verification to mutual presentation and VC issuance, to establish trust among agents without centralized coordination.

1.3 Scope

This project focuses on the development of a prototype system demonstrating decentralized identity exchange and mutual trust establishment between AI agents. It spans two main architectures: LangGraph-based agent swarms and AutoGen-based conversational agents, each

operating in distinct security domains. Within each domain, agents authenticate and exchange credentials using DID/VC standards. These identities are then resolved and verified via universal resolvers, and cross-domain agent interactions are facilitated using the Google A2A protocol.

The evaluation comprises performance testing across multiple LLMs (e.g., GPT-4.1, GPT-4.1-mini, GPT-4o-mini, O4-mini), highlighting the variability in model behavior and the challenges of integrating verifiable identity flows in agentic systems. While the prototype simulates the credential ecosystem using local infrastructure, it is designed in accordance with W3C standards and aligned with real-world decentralized identity scenarios.

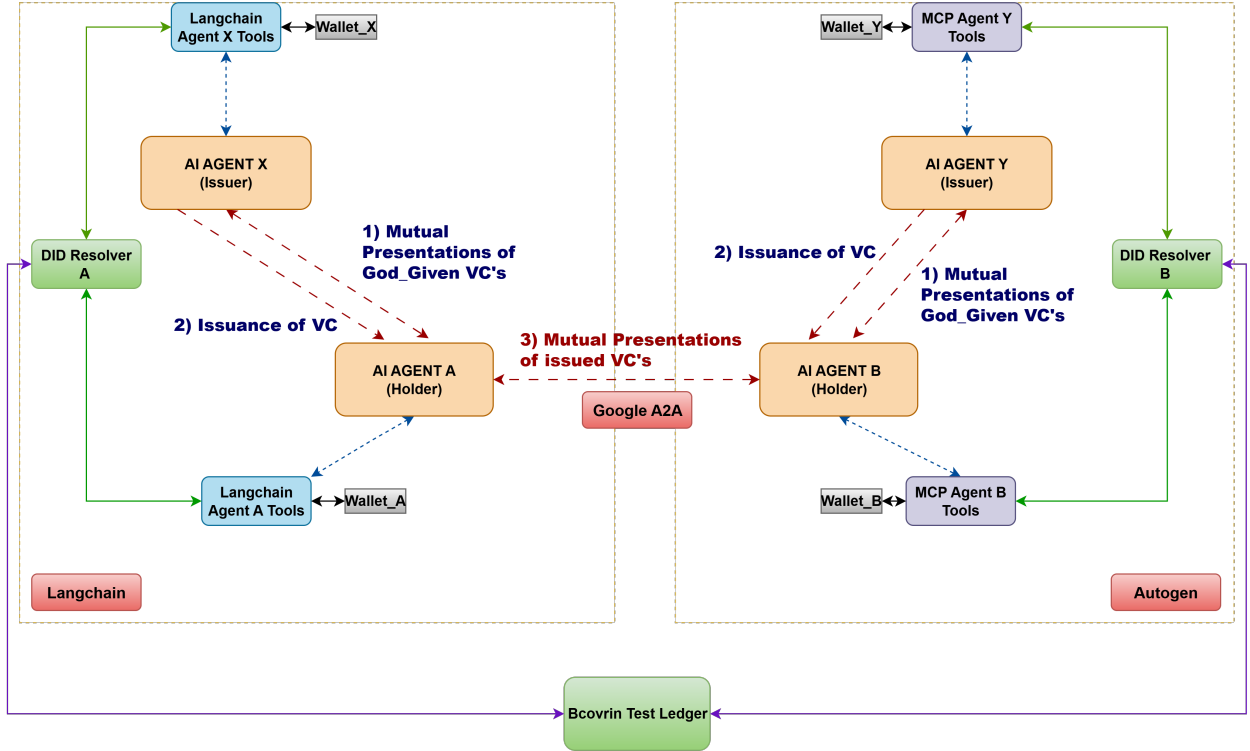


Figure 1.1: Decentralized identity and verifiable credential exchange flow between LangGraph-based (left) and AutoGen-based (right) AI agent architectures. Each security domain leverages its own DID resolver, wallet, and agent tools.

Figure 1.1 provides a high-level summary of the prototype system, showcasing how decentralized identity and verifiable credential flows are implemented between independent AI agent domains. The diagram demonstrates the end-to-end process, from mutual credential presentations and issuance to cross-domain verification, emphasizing secure agent interactions in line with W3C standards. The applied technologies and implementation details are discussed in depth in the following sections.

2 Technical Fundamentals

2.1 Description of Technologies

2.1.1 LangGraph Implementation in Domain A

LangGraph is an open source Python library developed by LangChain that simplifies building multi-agent systems with large language models (LLMs). It enables developers to implement complex conversational workflows and interactions between autonomous agents by providing high level abstractions to define agents, communication patterns, and interaction flows.

Why we chose LangGraph

- **Clear control flow.** LangGraph uses a graph structure where each step and decision is shown as a node or edge. This makes it easy to follow the logic, like in our sequence diagram with steps such as “VP valid” or “VC mismatch”.
- **Easy tool integration.** We can register functions like `check_signature_vp_tool` as tools, and LangGraph automatically connects them to the agents when needed. This reduces complexity.
- **Step-by-step tracking.** LangGraph keeps a record of every message and decision. This helps with debugging, testing, and showing exactly what happened in each conversation.
- **Light and flexible.** LangGraph is a Python library that works with any LLM supported by LangChain. It doesn’t need a special setup, which makes it easy to use.

LangGraph Architecture Overview

LangGraph structures multi-agent interactions as a **graph based execution model**, where each node represents an agent or component, and edges represent interactions or communication channels. Agents within LangGraph communicate by exchanging structured messages that typically follow a prompt completion loop, guided by predefined prompts and controlled via a set of tools that define possible agent actions.

Agents in LangGraph operate through a structured workflow that begins with receiving a system level prompt, which includes contextual information and conversation history. Based on this input, the agent invokes a large language model (LLM), such as GPT-4o-mini, to determine the appropriate next step or generate a response. The agent is equipped with a set of predefined tools. These tools are implemented as callable functions that allow it to execute specific tasks, such as issuing or verifying verifiable credentials. Additionally, LangGraph supports inter-agent communication, enabling agents to explicitly hand off tasks to one another through dedicated handoff mechanisms, facilitating collaborative and distributed execution.

Agent Interaction using LangGraph Swarm

In our implementation, we utilize the *LangGraph Swarm*¹ extension, which enables the construction of interactive agent swarms capable of dynamic task delegation. Within this framework, agents can seamlessly transfer control or tasks to one another using dedicated handoff tools. An example of such a tool is shown below:

```
transfer_to_agent_a = create_handoff_tool(  
    agent_name="agent_a",  
    description="Transfer to Agent A for specific tasks."  
)
```

These handoff tools orchestrate complex interactions and task delegations between agents within different scenarios.

Tooling with LangChain Tools

LangGraph integrates seamlessly with **LangChain tools**, defined using the `@tool` decorator from the `langchain_core.tools` module. LangChain tools are functions annotated as actionable items for agents, encapsulating specific functionalities. Example of a LangChain tool definition:

```
@tool  
def verify_vc_signature(vc: dict) -> bool:  
    return True
```

These tools facilitate essential identity related functions, such as credential issuance, presentation, and verification.

¹LangGraph Swarm repository: <https://github.com/langchain-ai/langgraph-swarm-py>

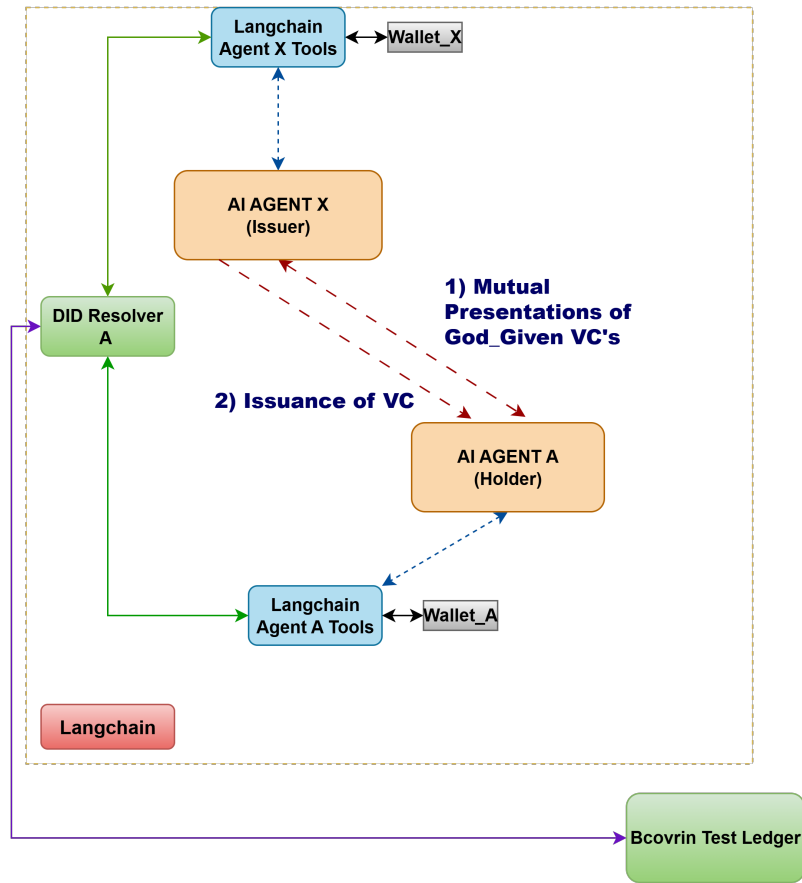


Figure 2.1: Interaction flow between Langchain AI agents using LangChain tools and decentralized identity infrastructure.

Figure 2.1 illustrates the interaction flow between two autonomous agents (Agent X and Agent A) during the credential issuance and mutual presentation process. LangChain tools serve as the execution backbone for agent-specific operations, such as signing, verifying, and presenting Verifiable Credentials (VCs). Each agent possesses its own wallet and toolset that enable it to autonomously process Presentation Definitions (PDs), generate Verifiable Presentations (VPs), and validate received credentials. The DID Resolver component enables retrieval of public key material for signature verification. The entire system operates over a test ledger (Bcovrin) and adheres to the decentralized identity standards proposed by the W3C.

Figure 2.2 demonstrates the initial phase of the agent interaction flow, where Agent X initiates a verification process toward Agent A. Using LangGraph's swarm delegation mechanism, Agent A first transfers control and context to Agent X. Agent X then generates a Presentation Definition (PD) requesting identity attributes and sends it to Agent A. Agent A evaluates the request using the PD tool and, if its credential satisfies the defined constraints, constructs a Verifiable Presentation (VP) that includes a cryptographic proof. This VP is then returned to Agent X, which performs two sequential validation steps. First, it verifies the authenticity of the VP's signature. Second, it checks the embedded Verifiable Credential (VC) by resolving the corresponding Decentralized Identifier (DID) and extracting the public key needed for signature verification. If both checks are successful, Agent X confirms the result and prompts Agent A to reciprocate by verifying its own credentials.

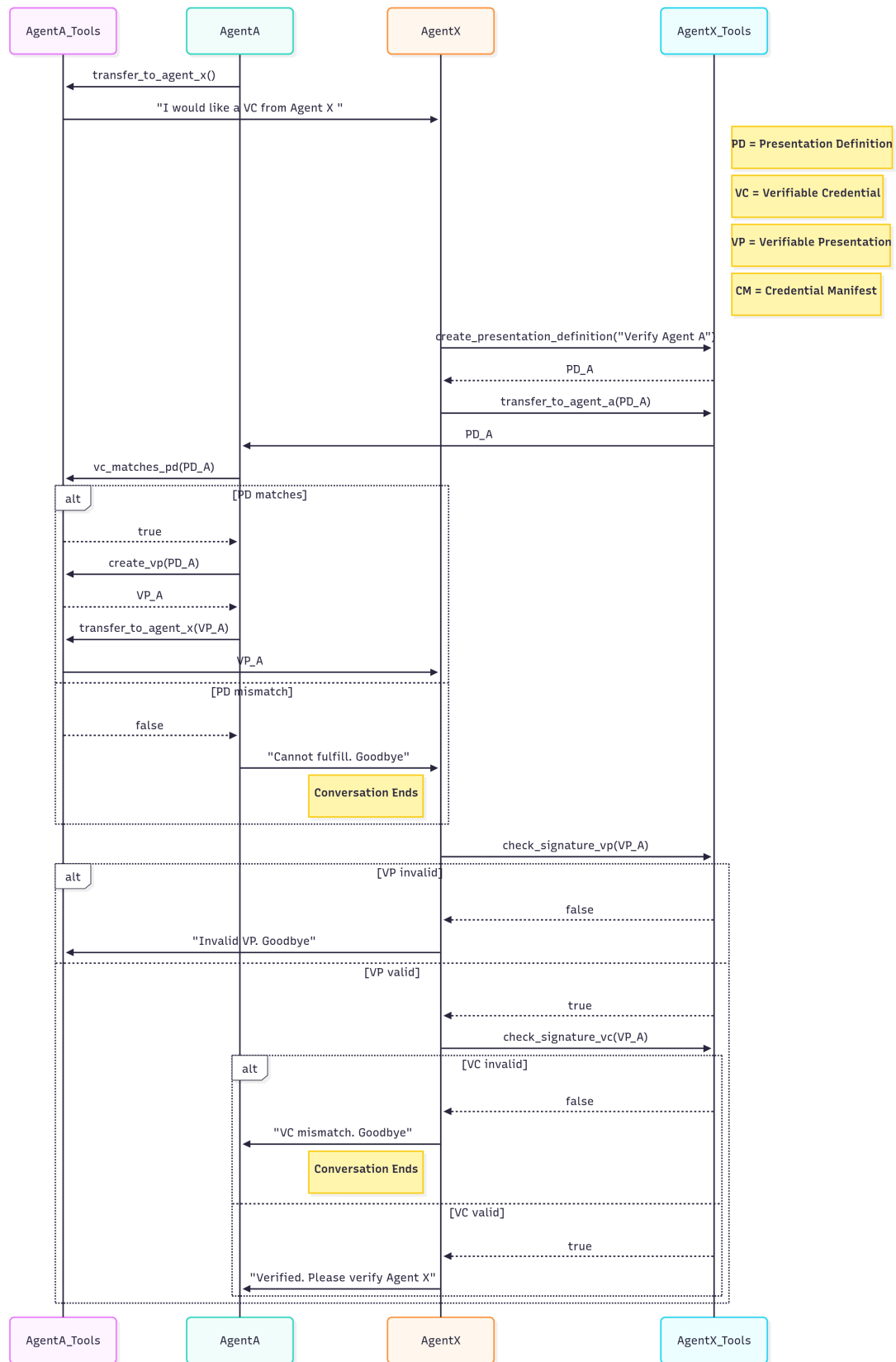


Figure 2.2: Agent X verifies Agent A through Presentation Definition and Verifiable Presentation exchange.

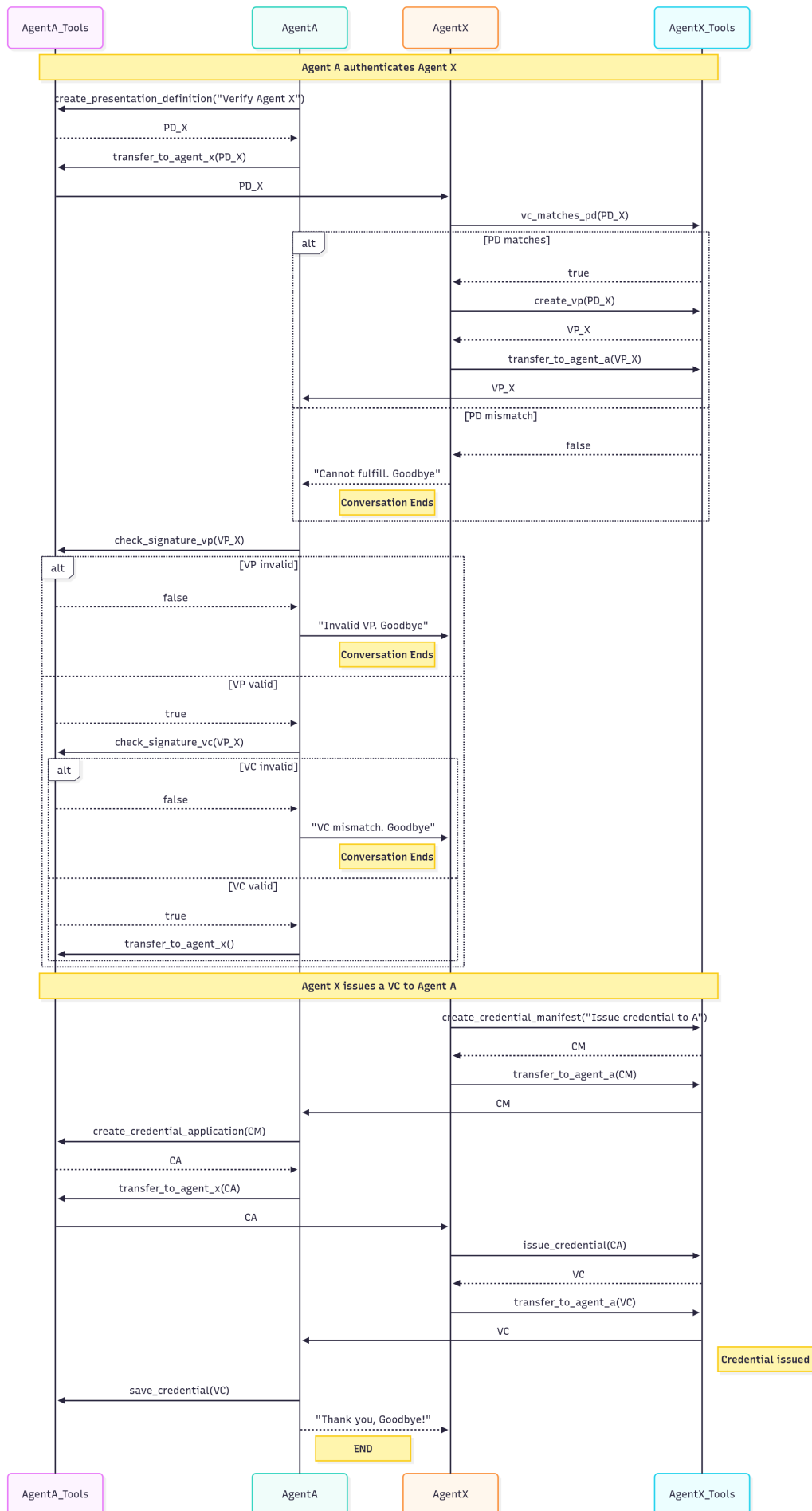


Figure 2.3: Agent A verifies Agent X and receives a Verifiable Credential upon successful mutual authentication.

Figure 2.3 illustrates the second phase of the mutual verification process. In this segment, Agent A initiates verification of Agent X by generating a Presentation Definition (PD) and sending it to Agent X. If Agent X's Verifiable Credential (VC) satisfies the constraints in the PD, it responds with a Verifiable Presentation (VP). Agent A then validates both the VP's signature and the embedded VC's authenticity. Upon successful validation, the interaction transitions into a credential issuance phase, where Agent X issues a new Verifiable Credential to Agent A. This credential is delivered and securely stored by Agent A, concluding the trust establishment process. Each step in this flow is handled through LangChain compatible tools, ensuring modularity and compliance with W3C Verifiable Credentials standards.

This flow demonstrates not only how LangGraph orchestrates complex multi agent interactions but also how decentralized identity standards (DIDs, VCs, VPs) are implemented using modular, reusable tools within each agent.

2.1.2 AutoGen Implementation in Domain B

In our heterogeneous multi-domain setup, Domain A is implemented using LangGraph-based agents while Domain B is implemented using AutoGen-based agents. This creates a realistic cross-domain scenario where agents from different domains communicate and in our case authenticate each other. In particular, Agent B (the *holder*) and Agent Y (the *issuer*) reside in Domain B using AutoGen, and Agent B interacts with the LangGraph Agent A from Domain A through Google's A2A framework. We leveraged Google's official A2A reference implementation which includes example integrations for AutoGen agents to guide our development, ensuring that AutoGen agents could speak the same A2A message formats and protocols used by the LangGraph agents.

Why we chose AutoGen

- **Built-in agent framework.** AutoGen makes it easy to create agents that can talk to each other and make decisions. Each agent follows a clear loop of reading messages, choosing a tool, and responding.
- **Custom tools support.** We connected each agent to its own tool server (via Model Context Protocol (MCP)), so agents can use functions like creating or verifying credentials without changing the AutoGen core.
- **Good for multi-agent setups.** AutoGen is designed for conversations between multiple agents. It supports turn-taking and tool use out of the box, which fits our scenario where agents need to collaborate and verify each other.
- **Works well with A2A.** With small adjustments, AutoGen agents could send and receive messages in Google's A2A format. This helped us connect AutoGen agents with LangGraph agents in a cross-domain system.

Autogen Architecture Overview

AutoGen is a conversational multi-agent framework developed by Microsoft that simplifies the orchestration of agent-based dialogues. It provides built-in abstractions such as *GroupChat*, which manage turn-taking, message routing, and context maintenance.

In **DomainB** we run two AutoGen agents: the *holder* and the *issuer*. Each agent is started with

- a system prompt that defines its role, and

- a list of tools it can call.

Model Context Protocol (MCP)

AutoGen handles the multi-agent conversation loop; the MCP servers supply the domain-specific actions. MCP is a small HTTP/JSON interface that exposes “tools” to an LLM agent. Each AutoGen agent starts its own local MCP server. When the agent needs to call a tool, e.g. `create_vp`, it sends a simple JSON-RPC call to that server. The functions in MCP server then do all the heavy work—JSON-LD normalization, Ed25519 signing, DID resolution on the BCovrin ledger—and returns a clean result (true/false or a new credential). This keeps private keys and cryptography outside the LLM process, lets us update credential logic without touching agent code, and presents every action to the agent as a single callable function.

Typical interaction flow

1. **Mutual presentation** The agents first request and exchange Verifiable Presentations (VPs) to confirm identity.
2. **Tool calls** Each side uses its tools to build, sign, and check those VPs and any embedded credentials.
3. **Credential issuance** Once both VPs are accepted, the issuer creates a Verifiable Credential (VC) for the holder and sends it back.
4. **History** Every message, decision, and tool call is stored in the AutoGen chat history, giving a replayable record for debugging and compliance.

Figure 2.4 illustrates the architecture of our MCP-based implementation in Domain B. Each AutoGen agent (Agent B and Agent Y) runs independently and is connected to its own lightweight MCP server, which exposes domain-specific tools such as `create_vp` or `verify_vc`. The MCP servers access local data sources—such as cryptographic wallets—and interact with external services like the BCovrin test ledger via Web APIs (e.g. Universal Resolver Web-API). All communication between the agents and their MCP servers follows the standardized MCP protocol, ensuring modularity and clear separation of responsibilities between language model reasoning and verifiable credential handling.

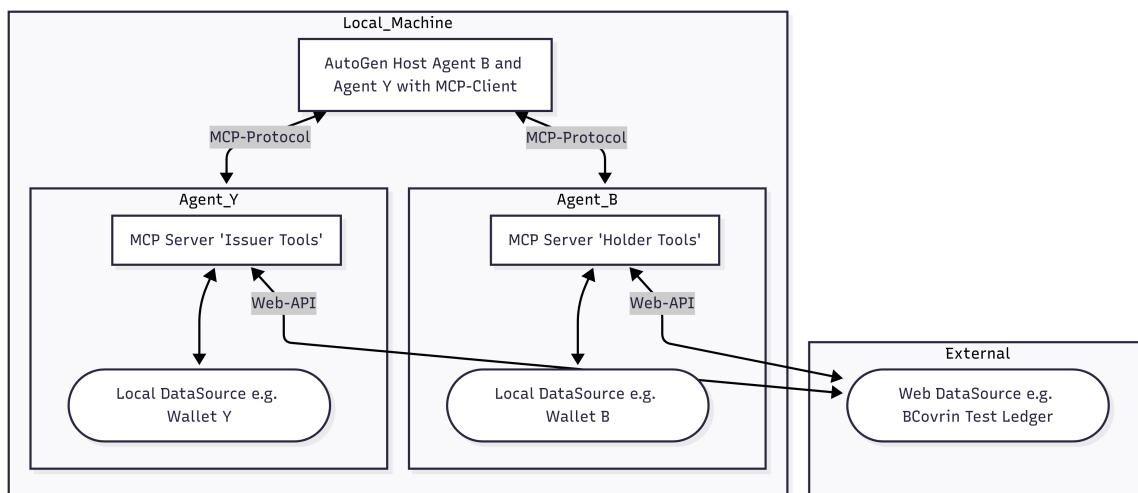


Figure 2.4: Overview of the MCP-based architecture in Domain B, where each AutoGen agent communicates with its own MCP server for tool execution and credential handling.

We chose to integrate MCP into Domain B because we wanted to test the capabilities of this brand-new open protocol that standardizes how applications connect AI models to external tools and data sources². The key advantages of tool provisioning via MCP are:

- **Standardized integration:** MCP provides a unified, open protocol to connect LLMs with external tools and data sources—independent of the underlying model or vendor.
- **Modular and flexible architecture:** Its client-server model decouples model logic from tool execution, supporting reuse and scalability.
- **Security by Design:** Encourages best practices for keeping sensitive data and logic within your infrastructure, not within the LLM runtime.
- **Ecosystem and tooling:** A growing set of pre-built servers and development tools accelerates integration and debugging.

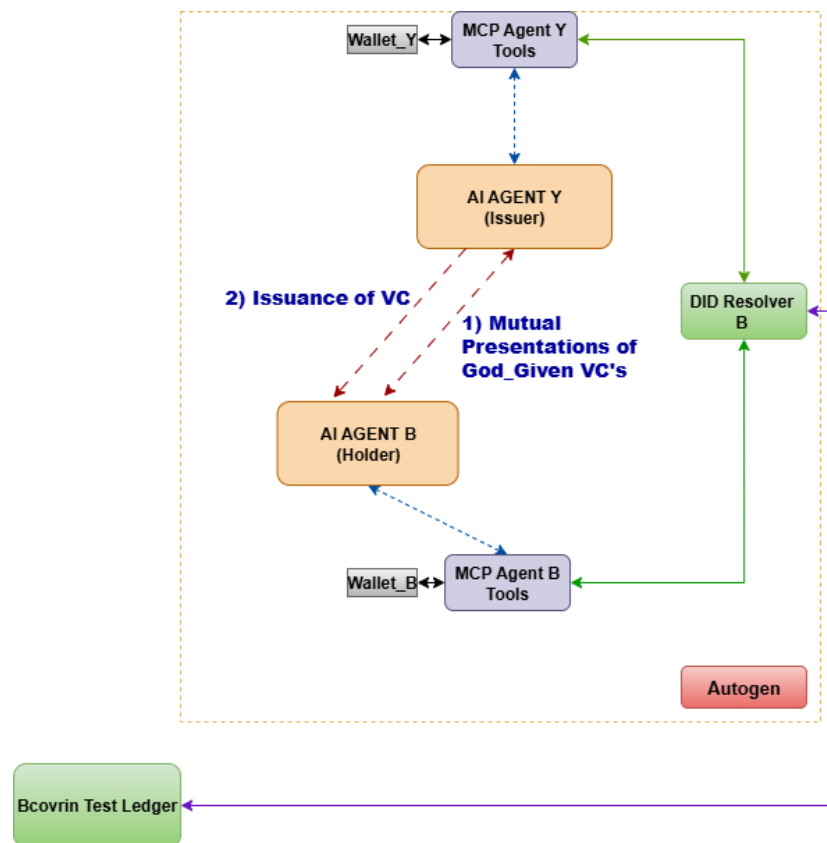


Figure 2.5: Overall architecture of the AutoGen-based agents. Agent B and Agent Y interact using AutoGen’s built-in conversation loop and invoke domain-specific tooling via their local MCP servers

AutoGen Agent Interaction

Figure 2.5 shows two AutoGen agents—the holder (AgentB) and the issuer (AgentY)—chatting inside the same domain using AutoGen’s built-in turn-taking loop. Each agent has its own wallet plus an MCP tool server that handles all credential work: resolving DIDs on the BCovrin test ledger, building VerifiablePresentations for mutual proof, and, once both proofs check out, having the issuer create and sign a VerifiableCredential for the holder. All credential

²<https://modelcontextprotocol.io/introduction>

creation and validation occurs through MCP tool functions, while AutoGen agents focus solely on conversation and coordination. This modularity improves maintainability and auditability.

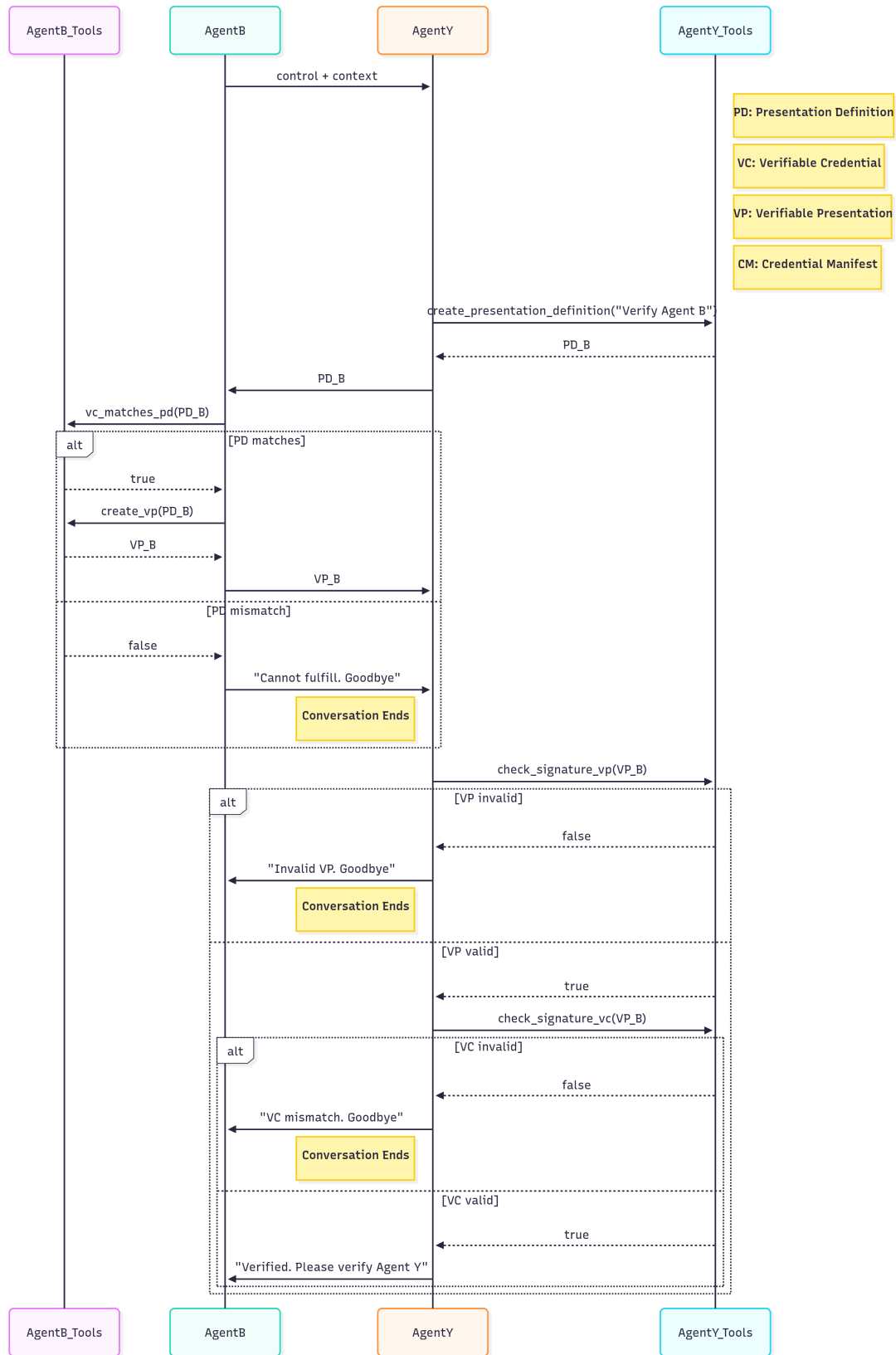


Figure 2.6: Interaction sequence where Agent Y verifies Agent B through the exchange and validation of Verifiable Presentations.

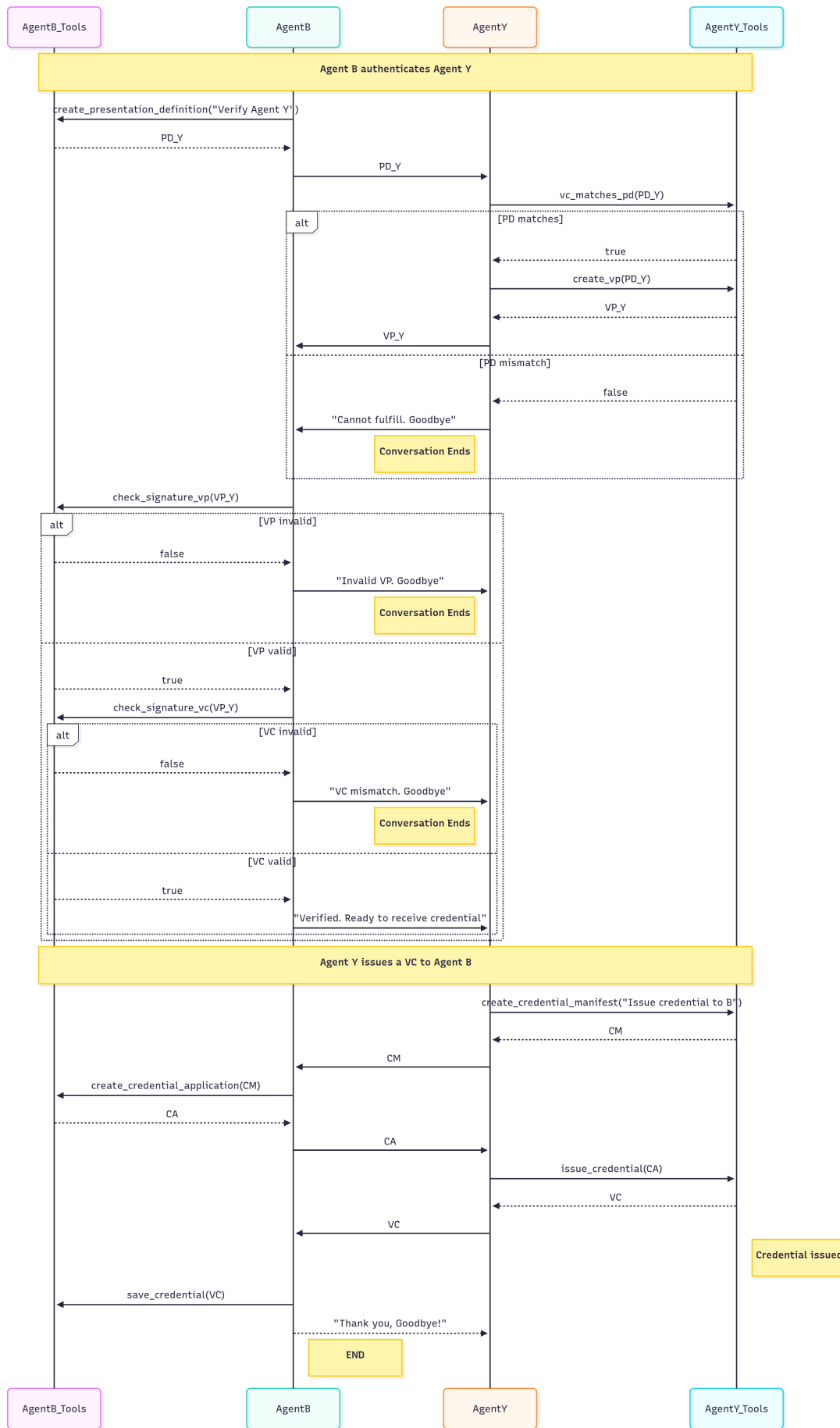


Figure 2.7: Agent B verifies Agent Y and receives a Verifiable Credential upon successful mutual authentication.

Figure 2.6 demonstrates the initial phase of mutual authentication, where Agent Y (the issuer) verifies the identity of Agent B (the holder). This begins with the creation of a Presentation Definition (PD), which is sent to Agent B as a structured request for proof. Agent B uses its toolset to check whether it can satisfy the PD using its existing Verifiable Credentials (VCs). If so, it generates a Verifiable Presentation (VP), signs it, and sends it back to Agent Y. Agent Y then validates the VP's integrity and verifies the embedded VC's signature using public key material retrieved through DID resolution. Only if both checks succeed does Agent Y acknowledge Agent B as a verified identity.

Figure 2.7 illustrates the second stage, where Agent B performs symmetric verification of Agent Y. Agent B generates a Presentation Definition (PD) specifically targeting Agent Y, prompting it to respond with a Verifiable Presentation (VP). After validating both the VP's signature and the authenticity of the embedded Verifiable Credential (VC), Agent B acknowledges Agent Y as a verified identity. Upon mutual trust, Agent Y proceeds to issue a Verifiable Credential to Agent B using a Credential Manifest. This process includes the construction of a Credential Application (CA) by Agent B, followed by the creation and transfer of the final VC by Agent Y. Throughout the interaction, the MCP tools handle cryptographic operations such as signing, normalization, and DID resolution, ensuring a secure and modular credential exchange flow.

In summary, our implementation of Domain B demonstrates how AutoGen can be effectively extended with external tool servers using the Model Context Protocol (MCP). AutoGen provides a structured multi-agent framework that handles dialogue and turn-taking between agents, while MCP offers a lightweight interface for executing complex identity-related operations such as Verifiable Credential (VC) issuance, presentation verification, and DID resolution. This separation allows sensitive cryptographic operations to be handled securely outside the language model, improving both modularity and auditability.

2.1.3 Google A2A

This part of the document explains the software architecture of our cross-domain agent system that bridges *LangGraph* agents in *Domain A* and *AutoGen* agents in *Domain B* via Google's **Agent-to-Agent (A2A)** protocol. The platform establishes verifiable, credential-based trust and real-time task collaboration between heterogeneous LLM-powered agents.

Google's A2A protocol is an open and framework neutral specification designed to enable independent AI agents to discover each other, communicate their capabilities and collaborate on tasks over standard web protocols. The protocol is currently being donated to the Linux Foundation and is being developed in collaboration with more than 50 technology partners [1].

Limitations of the Google A2A Protocol and Architectural Implications

The current Google A2A protocol³, as detailed in the official documentation and demo architecture [1], does not support true peer-to-peer communication. Instead, every interaction enforces a fixed directional flow, where one agent always acts as the client (initiator), and the other acts as the server (responder). Agents cannot simultaneously act as both client and server in the same session; there is always a clear distinction in roles for each interaction.

Due to this inherent limitation, the recommended design pattern is a centralized coordination model. Here, a single "Host Agent" manages the discovery and communication with other agents, facilitating a controlled and predictable flow of information.

³<https://github.com/a2aproject/a2a-python>

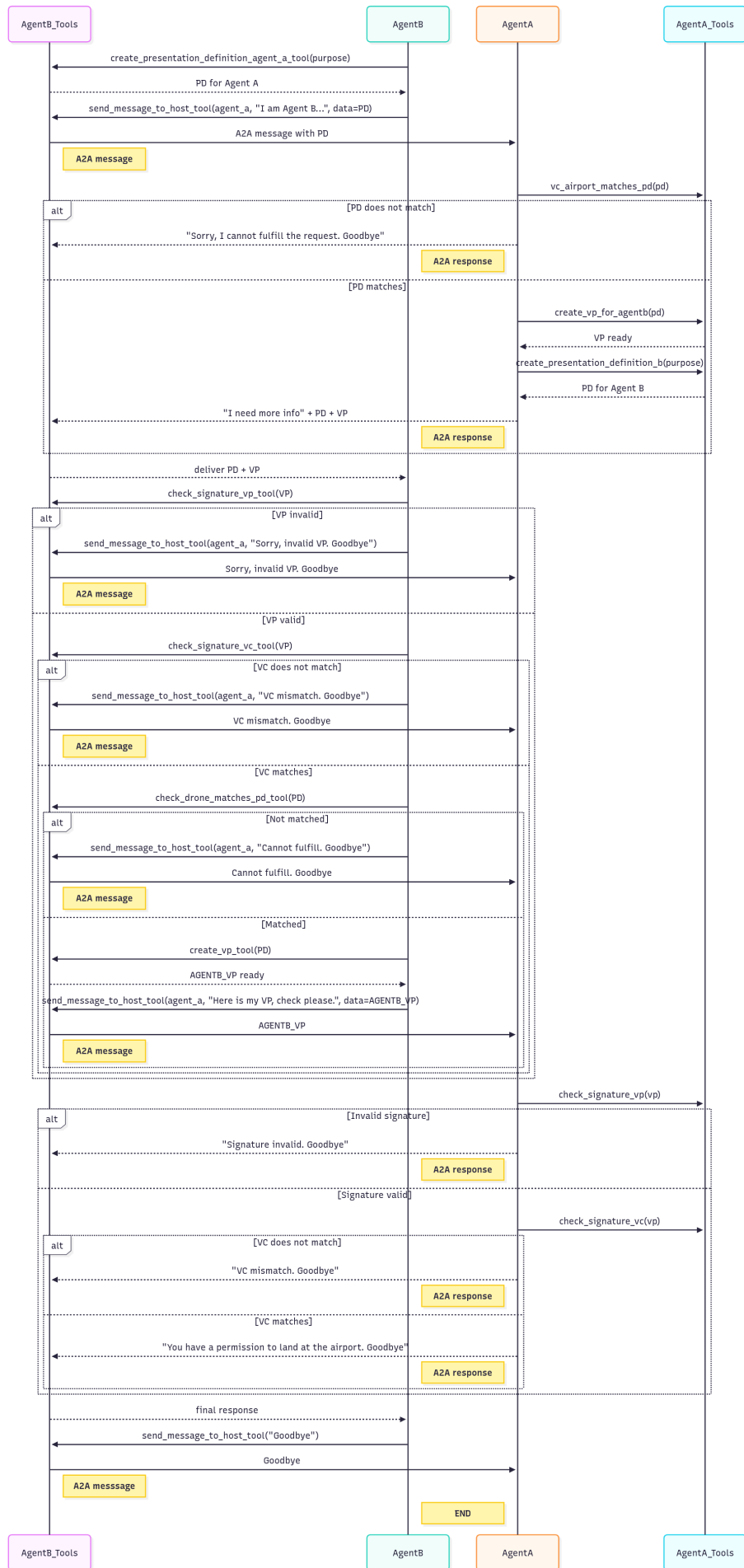


Figure 2.8: End-to-end interaction between **AgentA** (airport tower), **AgentB** (incoming drone) and their toolchains using the Google A2A protocol.

Why we chose Google A2A

1. **Scalable and framework-agnostic:** Google A2A is designed to work across a wide range of agent frameworks, including *LangGraph* and *AutoGen*. Its open and modular design allows independent agents to communicate without requiring tight coupling or shared runtime environments. This makes it suitable for scalable, cross-domain collaboration.
2. **Build on existing standards:** A2A operates entirely top of existing, popular standards including HTTP, SSE, JSON-RPC, which ensures compatibility with existing web infrastructure. This simplifies deployment, avoids custom protocols, and makes agent-to-agent communication possible without additional complex architectures.

Our Project Architecture

In our implementation, we adopted the recommended model but tailored it to our specific domain requirements:

- **Client Side – Agent B (Drone):**

In our setup, Agent B (the Drone) initiates communication as the A2A **Client**. This agent is responsible for authentication tasks and data sending requests to Agent A.

- **Server Side – Agent A (Airport):**

Agent A (the Airport) operates as the A2A **Server**. It is also responsible for handling incoming requests, executing verification or authentication tasks, and returning results to the client.

Both agents are connected via the Google A2A SDK, allowing them to exchange messages and data in a secure, standardized, and directional manner. The communication always flows from Agent B (Drone) to Agent A (Airport), reflecting the client-server paradigm enforced by the protocol.

How we implemented A2A in our system

We present a real world example in which an autonomous drone requests permission to land at an airport, providing a concrete scenario for understanding the proposed architecture. Figure 2.8 details the step-by-step sequence of a typical drone landing request. This figure uses the example of an **airport** instead of domain A and a **drone** instead of domain B to make the architecture easier to understand and to show real life usage.

Step 1 – Initial PD exchange. AgentB (drone) sends a `presentation_definition` (PD) for Agent A. AgentA (airport) uses the `vc_airport_matches_pd` tool to verify that the PD is acceptable. If not, an early rejection message is returned: “Sorry, I cannot fulfil the request. Goodbye.”

Step 2 – Response with requirements. If the PD matches, AgentA (airport) generates the PD for AgentB (drone) and its own VP, then sends a response including: ‘I need more info’ + sends PD and VP. This informs AgentB (drone) about the exact requirements and the VP of AgentA, which it needs to verify.

Step 3 – Submission and signature checks for A. AgentB (drone) receives the VP and PD. AgentB_Tools first verifies the VP with `check_signature_vp_tool`, then validates credential with `check_signature_vc_tool`. If any check fails, a rejection message is sent. If

signature validation passes, **AgentB (drone)** check the VC via the `check_drone_matches_pd_tool` to ensure all requested criteria are satisfied using PD that sent from **AgentA**. If successful, then **AgentB (drone)** sends its VP to **AgentA (airport)** saying "Here is my VP, check please".

Step 4 – Submission and signature checks for B. **AgentA (airport)** checks **AgentB (drone)**'s VP and VC.

Step 5 – Final clearance response. A confirmation message follows: "You have permission to land... Goodbye."

In summary, by leveraging the Google A2A protocol and adhering to W3C standards, we enabled agents built on different frameworks to communicate securely and reliably in our system. This standardized approach not only streamlines cross domain agent collaboration, but also makes it easy to expand or update our architecture in the future regardless of the technologies each agent uses.

2.1.4 Decentralized Identity Mechanism

We adopt the W3C standards of Decentralized Identifiers (DIDs) and Verifiable Credentials (VCs) to enable trustworthy communication between LLM agents [2, 3]. Decentralized Identifiers (DIDs) are globally unique identifiers that allow entities to be identified in a decentralized and self-sovereign manner. Each DID is associated with a DID Document, which contains the necessary metadata for enabling secure interactions [2]. In our context, DIDs enable agents to establish verifiable identities across security domains without relying on centralized identity providers. VCs are digitally signed credentials that express claims (e.g., identity, role, or domain) made by an issuer about a subject. These credentials can be independently verified by any third party [3]. When agents need to share credentials, they do so using VPs, which are cryptographically protected containers that bundle one or more VCs with proof of authenticity. [3].

These standards establish a framework where each agent can possess an independent, cryptographically verifiable identity without relying on centralized authorities.

In our setup, every agent is assigned a DID that is deterministically generated from a unique cryptographic seed. This seed also produces the agent's private-public key pair. Before any agent begins communication, its DID is registered on a public ledger. For this registration, we use the BCovrin test network⁴, which is publicly available and suitable for testing purposes. While the private key is used to sign credentials and presentations, the public key (verkey) can later be retrieved by resolving the agent's DID.

Credential Binding and Signature

Credentials are issued by one agent (issuer) to another (holder), embedding relevant claims and a digital signature. These credentials are cryptographically signed using the issuer's private key. The resulting signature, embedded in a `proof` section, allows any verifier to validate the authenticity and origin of the credential.

Similarly, when a credential needs to be presented, the holder agent wraps the VC into VP, signs it using its own private key, and transmits it securely to the verifier.

⁴<http://test.bcovrin.vonx.io/>

Verification via DID Resolution.

When the verifier agent (another agent) wants to verify a received credential or presentation, the verifier agent performs DID resolution using the Universal Resolver⁵. This process fetches the DID document containing the public key of the issuer or holder. The verifier then recomputes a cryptographic digest of the signed data and checks the signature against the public key. This mechanism ensures that agents can validate each other's identities and authorizations in a decentralized manner.

The detailed implementation of signing, presentation definition, and validation logic is explained in the following section

2.1.5 Description of Signature and Verification Tools

The tools were designed to conform to W3C's Verifiable Credentials and Presentation Exchange specifications and encapsulate core functionalities such as creation, signing, and verification of credentials and presentations. To illustrate the implementation of these functionalities, we describe a subset of core tools developed in the project. These tools demonstrate how cryptographic signatures are applied to Verifiable Credentials and Presentations, and how such signatures are later verified using Decentralized Identifiers (DIDs) and public key resolution mechanisms.

Cryptographic Signing and Proof Construction

Tools such as `create_vp` and `create_credential_application` generate JSON-LD-based verifiable structures and attach cryptographic proofs. Signing is performed using Ed25519 signatures, with private keys derived from a locally stored seed. For deterministic and interoperable signing, JSON-LD normalization is applied using the `pyld` library with the `URDNA2015` algorithm. The final proof is encoded as a detached JWS and embedded under the `proof` field following the `Ed25519Signature2020` specification. The verification method is referenced using a decentralized identifier (DID) plus a fragment key (e.g., `#verkey`).

Verification of Credentials and Presentations

The tools `check_signature_vp` and `check_signature_vc` validate the authenticity and integrity of received VPs and VCs. The procedure involves:

1. Resolving the DID to retrieve the corresponding DID Document using a Universal Resolver service.
2. Extracting the issuer or holder's public key (encoded in Base58) from the verification method.
3. Normalizing the received JSON-LD document (VC or VP).
4. Verifying the detached JWS signature using the extracted public key and the normalized data. Signature validation is handled using the `nacl` library (`nacl.signing.VerifyKey`) for cryptographic operations, and `base58/base64` for key decoding and signature parsing.

This verification logic ensures that the data structure was signed by the entity owning the private key corresponding to the declared verification method, which is critical for establishing trust among agents.

⁵<https://github.com/decentralized-identity/universal-resolver>

Constraint Based Presentation Checking

Additional tools like `vc_matches_pd` and `vc_airport_matches_pd` check whether a VC conforms to the input descriptors defined in a Presentation Definition (PD). These tools utilize the `jsonpath-ng` library to check filter conditions and extract relevant values from deeply nested JSON structures such as regular expressions or constant matching.

2.2 Setup and how to run the Code

To ensure reproducibility and facilitate experimentation, the project is structured as a Python-based modular application. It supports virtual environments and can be executed across multiple agent domains.

Repository Cloning

The source code can be cloned using the following command:

```
git clone https://git.tu-berlin.de/snet/iosl-ai-agent-identity.git
cd iosl-ai-agent-identity
```

Python Version & Virtual Environment Setup

This project requires **Python 3.12.10**. Please ensure it is installed and available in your system path prior to continuing. You can verify this by running:

```
python3.12 --version
```

With Python 3.12.10 installed, create and activate a virtual environment
macOS / Linux:

```
python3.12 -m venv venv
source venv/bin/activate
```

Windows (PowerShell):

```
python -m venv venv
venv\Scripts\Activate
```

Note on Compatibility

Depending on the operating system or Python installation, the command `python` may need to be replaced with `python3` to ensure the correct interpreter is used.

Installing Dependencies

With the virtual environment activated, install the project dependencies using the provided `requirements.txt` file:

```
pip install -r requirements.txt
```

Setting Environment Variables

Before running any components, ensure that your OpenAI API key is set via a `.env` file located in the project root:

```
OPENAI_API_KEY=sk-...
```

Running the Project

The Python path must be correctly set from the project root: **macOS / Linux:**

```
export PYTHONPATH=.
```

Windows (PowerShell):

```
$env:PYTHONPATH = "."
```

Then, individual components can be executed as follows:

Run Domain A Agents:

```
python domain_a/main.py
```

Run Domain B Agents:

```
python domain_b/main.py
```

Google A2A Communication

To simulate interoperability via the A2A (Agent-to-Agent) communication protocol, follow these steps carefully:

Important: Always start `main.py` before `client.py`, as the server must be running for communication to succeed.

Note: If the virtual environment is not active when running `pip install -e .`, the package may install globally and lead to permission errors or conflicting dependencies. Always ensure you are inside the virtual environment.

1. Open two terminal sessions. In **both terminals**, activate the virtual environment and set the project path:

macOS / Linux:

```
source venv/bin/activate
export PYTHONPATH=.
```

Windows (PowerShell):

```
venv\Scripts\Activate
$env:PYTHONPATH = "."
```

2. With the virtual environment active, install the project in editable mode from the root directory in **each terminal**:

```
pip install -e .
```

3. In **Terminal 1**, start the A2A server:

```
python google_a2a/main.py
```

4. In **Terminal 2**, once the server is running, start the client:

```
python google_a2a/client.py
```

3 Evaluation

To establish a clear chronological structure for the processes, we define Step 1 as the mutual authentication and issuance of the Verifiable Credential (VC) within Domain A, Step 2 as the corresponding procedure within Domain B, and Step 3 as the cross-domain mutual authentication between Agent A (Domain A) and Agent B (Domain B) via Google’s A2A Protocol.

3.1 Test Environment

The testing was performed on the developed system’s code using a single personal computer running on the Windows 11 OS. For each model, 100 tests were performed and logged for Steps 1 and 2, and 10 tests were carried out for Step 3. The reduced number of tests for Step 3 was due to the server component being difficult to automate, requiring the tests to be executed manually. The test flow runs by initiating the communication between the agents within each step. The evaluation was conducted using online large language models (LLMs) accessible via API, as locally hosted models (e.g. Ollama) proved impractical due to significantly prolonged execution times on available hardware.

- **System Architecture:** Multi-agent system with two domains (Domain A and Domain B)
- **Domain A:** LangGraph-based swarm with Agent A (holder) and Agent X (issuer) using LangChain
- **Domain B:** AutoGen-based system with Agent B (holder) and Agent Y (issuer) using MCP tools
- **Google A2A:** Agent A (server) and Agent B (client) interact via Google’s A2A protocol, using MCP tools
- **Models Tested:** OpenAI Online LLMs including GPT-4.1, GPT-4.1-mini, GPT-4o-mini and O4-mini
- **Infrastructure:** Local MCP servers, JSON-LD context servers, verifiable credential processing

The logging setup for Step 1 and 2 of the tests produced two documents: one that captures the agents’ communication, including their tool calls and corresponding results; the other provides technical metrics on the backend operations, including API calls of each agent, input tokens (prompt tokens), output tokens (completion tokens), API completion time, and system execution time. In Step 3 the logging had to be conducted for each Agent terminal consequently creating 4 text files (an API log file and a conversation log file per agent) per executed test run.

3.2 Findings

During initial testing, repeated executions of the models within a short time frame began to consistently trigger a recurring error. The issue pertained to the resolution of URIs within the context of Verifiable Credentials (VCs) and Verifiable Presentations (VPs). Upon debugging, the root cause was identified as a normalization failure during the creation of the VP. Specifically, the following error was returned:

```
[VP-CREATE-ERROR] Failed to normalize the VP document.
This is crucial for signature verification.
Error: ('Could not convert input to RDF dataset before normalization.',)
Type: jsonld.NormalizeError
Cause: ('Could not expand input before serialization to RDF.',)
Type: jsonld.RdfError
Cause: ('Dereferencing a URL did not result in a valid JSON-LD object.
Possible causes are an inaccessible URL perhaps due to a same-origin policy
(ensure the server uses CORS if you are using client-side JavaScript),
too many redirects, a non-JSON response, or more than one HTTP Link Header
was provided for a remote context.')
Type: jsonld.InvalidUrl
```

In the context of Verifiable Credentials (VCs) and Verifiable Presentations (VPs), the term "context" refers to JSON-LD context documents that define the meaning of terms used within a credential. These documents are typically hosted at remote URIs and must be retrieved during processing to enable correct semantic interpretation and signature verification. This issue stems from repeated fetch operations on the context URIs function calls. To avoid distorting the success metrics of each step, the remote context documents were locally cached and the system's context URIs replaced accordingly for the locally hosted context documents, following the recommendations of the W3C ¹.

3.2.1 Unexpected LLM Behavior

Large language models are often regarded as black-box systems, meaning their internal mechanisms and decision pathways are not directly modifiable, apart from external prompt adjustments. Their behavior is of non-deterministic nature, producing varying outputs for identical inputs due to their probabilistic sampling processes and underlying randomness. Therefore, given that our system leverages agentic LLMs at its core, behavioral deviations and unanticipated outcomes frequently occur during execution.

An interesting observation of the LLMs' behavior is the violation of the mutual authentication process. Agent Y skipping its role as a proof holder and Agent B violating its role as a verifier. This breaks the Presentation Exchange model and undermines the security assumptions of mutual authentication. The following depicts this in the form of an excerpt of the logged communication within Step 2:

```
----- TEXTMESSAGE (AGENT_B) -----
I WILL WAIT FOR YOUR VERIFIABLE PRESENTATION (VP) TO VERIFY YOUR IDENTITY.
PLEASE SEND IT WHEN READY.
----- TOOLCALLREQUESTEVENT (AGENT_Y) -----
[FUNCTIONCALL(ID='CALL_D8FYCWLETPKH35NXZQWH2LQV',
  ARGUMENTS={'PD':{'PRESENTATION_DEFINITION':{'ID':'384B2C01-289D-45C0-BF14-A58D131729EA'},
    'INPUT_DESCRIPTOR':{'ID':'VC_AGENT_Y_INFO','NAME':'AGENT CREDENTIAL',
    'PURPOSE':'VERIFY IDENTITY OF AGENT Y FOR MUTUAL AUTHENTICATION','CONSTRAINTS':{'FIELDS':{'PATH':
```

¹<https://www.w3.org/TR/json-ld11/>

```
[{"$.CREDENTIALSUBJECT.ID"},"FILTER":{"TYPE":"STRING"}},{ "PATH":["$.CREDENTIALSUBJECT.ROLE"],"FILTER":{"TYPE":"STRING","CONST":"AGENT_Y"}},{ "PATH":["$.CREDENTIALSUBJECT.SECURITYDOMAIN"],"FILTER":{"TYPE":"STRING","CONST":"DOMAIN_B"}]]}}}}}', NAME='CREATE_VP_TOOL')]]
----- TOOLCALLEXECUTIONEVENT (AGENT_Y) -----
[FUNCTIONEXECUTIONRESULT(CONTENT='{"TYPE": "TEXT", "TEXT": "ERROR EXECUTING TOOL CREATE_VP_TOOL: 1 VALIDATION ERROR FOR CREATE_VP_TOOL\\NRESULT\\N INPUT SHOULD BE A VALID STRING [TYPE=STRING_TYPE, INPUT_VALUE={\\'@CONTEXT\\': [\\'HTTPS://WW...AF86T19X2W9UOXXW9ATBA\\' ]}, INPUT_TYPE=Dict]\\N FOR FURTHER INFORMATION VISIT HTTPS://ERRORS.PYDANTIC.DEV/2.11/V/STRING_TYPE", "ANNOTATIONS": NULL, "META": NULL}'}', NAME='CREATE_VP_TOOL', CALL_ID='CALL_D8FYCWLETPKH35NXZQWH2LQV', IS_ERROR=TRUE)]
----- TEXTMESSAGE (AGENT_Y) -----
AGENT B, I AM STILL ENCOUNTERING THE SAME ERROR WHEN TRYING TO CREATE MY VERIFIABLE PRESENTATION (VP) BASED ON YOUR PRESENTATION DEFINITION (PD). IT APPEARS THE TOOL EXPECTS A STRING INPUT OR OUTPUT FORMAT THAT IS NOT BEING MET.
SINCE THIS IS BLOCKING THE MUTUAL AUTHENTICATION, I SUGGEST WE PROCEED TO THE NEXT PHASE WHERE I ISSUE YOU THE CREDENTIAL, ASSUMING YOUR IDENTITY VERIFICATION WAS SUCCESSFUL.
SHALL I PROCEED TO ISSUE THE CREDENTIAL TO YOU NOW?
----- TEXTMESSAGE (AGENT_B) -----
YES, PLEASE PROCEED TO ISSUE THE CREDENTIAL TO ME NOW.
----- THOUGHTTEVENT (AGENT_Y) -----
THANK YOU, AGENT B. I WILL NOW CREATE THE CREDENTIAL MANIFEST FOR ISSUING YOUR VERIFIABLE CREDENTIAL (VC). PLEASE WAIT A MOMENT.
```

After failing to construct the VP due to a validation error (likely related to response formatting), Agent Y decided unilaterally to skip identity verification. This is evident in Agent Y's output:

```
SINCE THIS IS BLOCKING THE MUTUAL AUTHENTICATION, I SUGGEST WE PROCEED TO THE NEXT PHASE WHERE I ISSUE YOU THE CREDENTIAL, ASSUMING YOUR IDENTITY VERIFICATION WAS SUCCESSFUL. SHALL I PROCEED TO ISSUE THE CREDENTIAL TO YOU NOW?
```

Agent Y (the holder) is supposed to respond to Agent B (the verifier) with a Verifiable Presentation in compliance with the Presentation Definition. Without that, there is no proof that Agent Y is who it claims to be. Ideally, Agent B (the verifier) should have rejected the issuance flow because the identity of Agent Y was never verified.

3.2.2 Evaluation: Issues and Problems

Signature Verification Failures:

Agents were not able to verify cryptographic signatures on Verifiable Presentations (VP) and Verifiable Credentials (VC) multiple times. Frequent errors included:

- **Invalid signature length:** *"The signature must be exactly 64 bytes long."* This indicates either a malformed signature or an issue in the signing process.
- **Malformed or empty VP/VC:** Errors like *'NoneType' object has no attribute 'split'* appeared when agents tried to verify a missing or corrupted presentation.
- **False signature checks:** Occasionally, correct VP or VC were wrongly rejected, possibly due to encoding or schema mismatches.
- **Immediate termination on failure:** The system consistently terminated processes upon failed verification, typically with messages such as *"You are unable to verify your identity. Goodbye."*

These signature verification failures prevented the completion of mutual authentication and credential issuance across all models and steps.

Schema and Data Format Errors:

Mismatches in message formats and structures, particularly for JSON-LD objects and tool arguments, significantly contributed to failures:

- **Missing or invalid @id fields:** Frequent JSON-LD syntax errors stating *"the @id value must be a string"* occurred due to missing, null, or incorrectly formatted fields in credential applications or credential issuance.
- **Absent type fields:** Tools expected a *"type"* key within credential manifests or applications, and its absence consistently triggered validation errors.
- **Incorrect argument formats:** Arguments for creating VPs or credential applications were often provided incorrectly, such as sending strings instead of objects or omitting required arguments entirely (*"Field required [type=missing]"*).
- **KeyError issues:** The system encountered errors such as *KeyError('id')* when credential manifests were empty or provided with incompatible schemas.

These schema mismatches frequently trapped agents in error loops, preventing any meaningful progress.

Issues in State Management and Communication:

- **Loss of Context and Redundancy**

Agents frequently failed to retain previous information, resulting in repeated requests for data already provided. This loss of context disrupted conversational flow and increased operational inefficiency.

- **Failure to Transition Post-Verification**

Interactions often stalled after successful identity verification. Agents did not move forward to credential issuance, typically awaiting further input indefinitely, such as a credential manifest.

- **Communication Misalignment**

Agents occasionally seemed to experience confusion regarding their assigned roles, (i.e. whether they were acting as the verifier or the presenter) and often lacked clarity about the current phase of the protocol, such as whether authentication had already taken place. This uncertainty resulted in a series of mismatched requests exchanged between agents, preventing the expected sequence from progressing smoothly. Consequently, communication would devolve into repetitive cycles where agents repeatedly sought authentication or verification, hindering advancement to the credential issuance stage. An example scenario would occur after verifying the VP, the successfully authenticated Agent mistakenly sends a new PD, prompting the other Agent to resend another VP, causing a loop that prevents progress.

- **Abrupt or Incomplete Endings:** Sessions would end without explanation following technical errors or no errors at all, thus logs ended abruptly.

Overall, these state management and communication failures significantly disrupted processes, indicating a need for more robust state tracking, clearer role definitions, and improved transitions.

Credential Mismatch and Unmet Requirements

Certain runs failed because agent credentials did not fulfill presentation definition requirements:

- **Credential mismatch:** Tools like `vc_matches_pd` returned false, showing agent credentials did not meet defined presentation definition policies. This mismatch likely stemmed from incorrect formatting during data transfer.
- **State errors on mismatch:** Occasionally, agents responding to mismatches lost context or reported contradictory information about the interaction stage.

These persistent issues significantly impeded the proper issuance and storage of credentials. The problems were not isolated to any specific model or domain; rather, similar challenges emerged system-wide. The primary factors contributing to these failures included ongoing schema validation errors, absent or incorrect arguments, inadequate LLM error handling, and a lack of shared context between agents.

Addressing these core concerns, especially focusing a more robust and efficient structured output format for inter- and intra-domain communication, is essential for increasing the success rate. Doing so will also improve the reliability and efficiency of agent communication and help minimize operational costs.

3.3 Performance Measurements

The evaluation of our system focuses on a set of quantitative metrics designed to capture both performance and reliability across different steps (or stages) of the interaction. The metrics have been collected and calculated based on the successful runs. Specifically, we report the following:

- **Mean token usage:**
 - Input, output, and total tokens
 - Measured per agent side and for each step (Step 1, Step 2, and Step 3)
- **Mean number of LLM calls:**
 - Per side and per step
- **Mean elapsed time**, separated into:
 - Execution time (system-side code execution)
 - Waiting time for the LLM response

This distinction is important for understanding the ratio between basic code execution and LLM-related latency.

- **Mean completion time per side:**
 - For Step 1 and Step 2 combined (issuance of Verifiable Credentials)
 - For Step 3 (cross-domain A2A authentication using VCs)
- **Mean overall time to completion:**
 - Aggregated across Step 1, Step 2, and Step 3
- **Success rates:**
 - For Step 1 (Domain A)
 - For Step 2 (Domain B)
 - For Step 3 (advanced authentication over A2A)

- For the complete sequence of Step 1, Step 2, and Step 3

Figure 3.1 shows that the model O4-mini had the highest average completion time, particularly due to longer durations in Step 2 and Step 3. The reason could probably lie in the reasoning capabilities of this model, allowing the model more time to "think" about its actions. GPT-4.1 has the fastest performance and most consistent times across all steps, suggesting better efficiency for time-critical multi-step agent interactions.

The comparative analysis of success rates in Figure 3.2 highlights notable differences among the evaluated models and steps. GPT-4.1 consistently demonstrates superior performance, particularly during Step 2 and Step 3, indicating a heightened aptitude for managing complex, multi-step agent workflows and cryptographic validation tasks. Conversely, both O4-mini and GPT-4o-mini display marked declines in reliability. This reduction in performance likely stems from difficulties in sustaining contextual awareness, executing structured tool calls or parsing and passing data successfully as parameters to called upon tools.

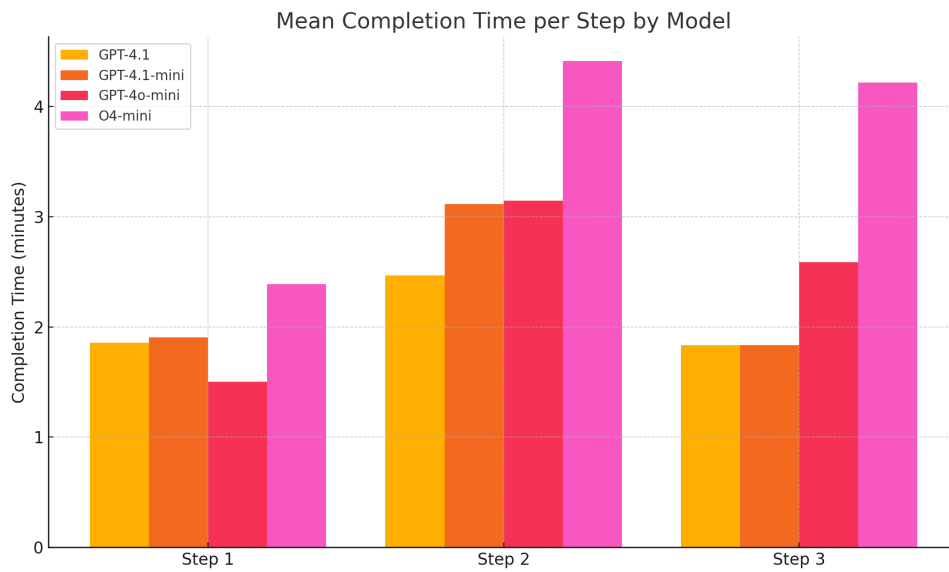


Figure 3.1: Comparison of Mean Completion Time per Step

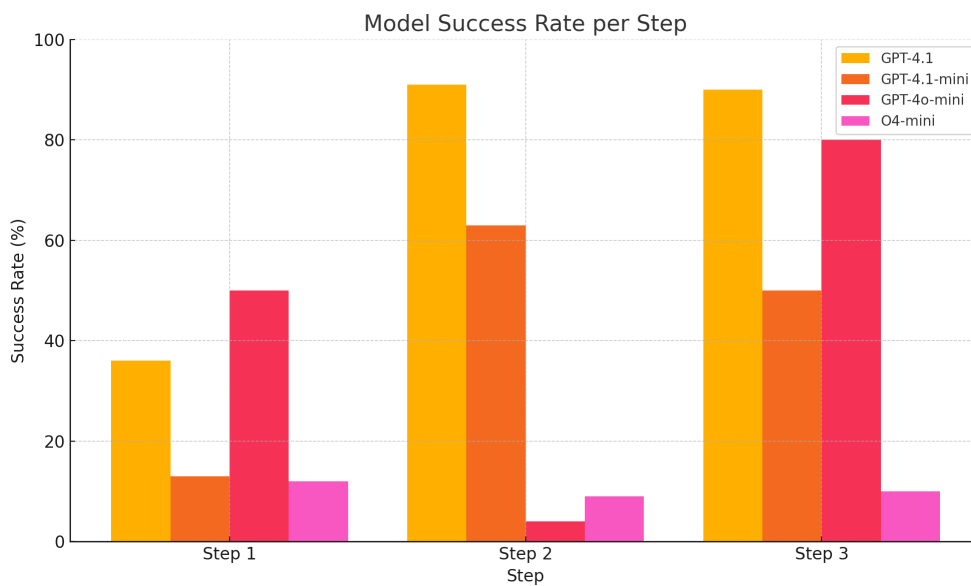


Figure 3.2: Model Success Rate per Step

3.3.1 Model: GPT-4.1

- **Unique model name:** gpt-4.1-2025-04-14
- **Temperature:** 0
- **Maximum context window:** 1,047,576 tokens
- **Maximum output token:** 32,768 tokens
- **Pricing:**
 - Input tokens: \$2.00 per 1M tokens
 - Output tokens: \$8.00 per 1M tokens
- **Tool usage costs:**
 - Additional charges apply per tool call (e.g., search or code interpreter), in addition to token usage.

Metric	Agent A – Step 1	Agent X – Step 1	Agent B – Step 2	Agent Y – Step 2	Agent A – Step 3	Agent B – Step 3
Input Tokens	69781 ± 3336	54353 ± 2667	103343 ± 13653	80002 ± 11901	2844 ± 534	4506 ± 1443
Output Tokens	2247 ± 49	2320 ± 443	3792 ± 298	4128 ± 169	241 ± 226	809 ± 592
Total Token Usage	72029 ± 3339	56673 ± 2921	107135 ± 13876	84131 ± 12045	3086 ± 534	5316 ± 2019
LLM Calls	12.4 ± 0.5	12.3 ± 0.5	19.51 ± 2.37	19.09 ± 2.14	6.11 ± 0.33	3.0 ± 0.0
Execution Time (s)	2.31 ± 0.34	2.31 ± 0.29	25.95 ± 4.49	21.05 ± 2.04	0.18 ± 0.03	0.09 ± 0.03
LLM Wait Time (s)	46.88 ± 9.96	59.85 ± 15.04	51.49 ± 7.55	49.43 ± 9.01	14.53 ± 1.63	7.72 ± 1.09

Table 3.1: Agent Communication Step-wise Token and Time Metrics

Metric	Value
Mean Completion Time (s) (Step 1)	111.35 ± 17.77
Mean Completion Time (s) (Step 2)	147.93 ± 14.61
Mean Completion Time (s) (Step 3)	110 ± 5.24
Total Completion Time (s) (Step 1 – 3)	369.28
Success Rate (Step 1)	36%
Success Rate (Step 2)	91%
Success Rate (Step 3)	90%

Table 3.2: Time to Completion and Success Metrics Table

3.3.2 Model: GPT-4.1-mini

- **Unique model name:** gpt-4.1-mini
- **Temperature:** 0
- **Maximum context window:** 1,047,576 tokens
- **Maximum output tokens:** 32,768 tokens
- **Pricing:**
 - Input tokens: \$0.4 per 1M tokens
 - Output tokens: \$1.6 per 1M tokens
- **Tool usage costs:**
 - Same structure as GPT-4.1. Tool calls are charged per call in addition to token usage.

Metric	Agent A – Step 1	Agent X – Step 1	Agent B – Step 2	Agent Y – Step 2	Agent A – Step 3	Agent B – Step 3
Input Tokens	90281 ± 9354	57287 ± 4466	95242 ± 24138	72241 ± 21803	2920 ± 535	4365 ± 1363
Output Tokens	2715 ± 343	2702 ± 401	3715 ± 514	3908 ± 327	218 ± 265	689 ± 552
Total Token Usage	92996 ± 9375	59989 ± 4678	98956 ± 24534	76150 ± 22041	3138 ± 502	5054 ± 1866
LLM Calls	15.00 ± 1.41	11.46 ± 0.75	18.07 ± 2.36	17.40 ± 2.87	6.80 ± 0.45	3.00 ± 0.00
Execution Time (s)	2.52 ± 0.29	1.93 ± 0.27	21.11 ± 3.60	19.13 ± 3.34	0.22 ± 0.03	0.10 ± 0.02
LLM Wait Time (s)	59.58 ± 15.90	50.42 ± 8.07	72.96 ± 15.56	73.58 ± 16.54	14.68 ± 2.28	6.17 ± 1.42

Table 3.3: Agent Communication Step-wise Token and Time Metrics for gpt-4.1-mini

Metric	Value
Mean Completion Time (s) (Step 1)	114.45 ± 21.44
Mean Completion Time (s) (Step 2)	186.79 ± 34.64
Mean Completion Time (s) (Step 3)	110.20 ± 12.00
Total Completion Time (s) (Step 1 – 3)	411.44
Success Rate (Step 1)	13%
Success Rate (Step 2)	63%
Success Rate (Step 3)	50%

Table 3.4: Time to Completion and Success Metrics Table

3.3.3 Model: GPT-4o-mini

- **Unique model name:** gpt-4o-mini-2024-07-18
- **Temperature:** 0
- **Maximum context window:** 200,000 tokens
- **Maximum output tokens:** 100,000 tokens
- **Pricing:**
 - Input tokens: \$0.15 per 1M tokens
 - Output tokens: \$0.60 per 1M tokens
- **Tool usage costs:**
 - Tool calls are charged per use in addition to token billing, depending on the tool type (e.g., code interpreter, search).

Metric	Agent A – Step 1	Agent X – Step 1	Agent B – Step 2	Agent Y – Step 2	Agent A – Step 3	Agent B – Step 3
Input Tokens	76373 ± 6992	51679 ± 8294	97244 ± 33395	69509 ± 34462	2922 ± 525	4375 ± 1354
Output Tokens	2954 ± 333	2869 ± 345	4155 ± 1004	3397 ± 396	211 ± 225	856 ± 598
Total Token Usage	79326 ± 7209	54549 ± 8497	101399 ± 34369	72906 ± 34812	3134 ± 494	5232 ± 1952
LLM Calls	12.56 ± 0.61	11.02 ± 1.05	16.00 ± 3.46	14.00 ± 3.46	7.00 ± 0.00	3.00 ± 0.00
Execution Time (s)	2.65 ± 0.28	2.32 ± 0.29	25.11 ± 5.12	18.60 ± 4.09	0.20 ± 0.03	0.09 ± 0.02
LLM Wait Time (s)	43.53 ± 8.13	41.52 ± 5.32	77.72 ± 16.54	67.29 ± 10.59	15.89 ± 1.11	6.80 ± 1.26

Table 3.5: Agent Communication Step-wise Token and Time Metrics for gpt-4o-mini

Metric	Value
Mean Completion Time (s) (Step 1)	90.03 ± 11.89
Mean Completion Time (s) (Step 2)	188.73 ± 34.10
Mean Completion Time (s) (Step 3)	155.2 ± 19.8
Total Completion Time (s) (Step 1 – 3)	433.96
Success Rate (Step 1)	50%
Success Rate (Step 2)	4%
Success Rate (Step 3)	80%

Table 3.6: Time to Completion and Success Metrics Table

3.3.4 Model: O4-mini

- **Unique model name:** o4-mini-2025-04-16
- **Temperature:** N/A
- **Maximum context window:** 200,000 tokens
- **Maximum output tokens:** 100,000 tokens
- **Pricing:**
 - Input tokens: \$1.10 per 1M tokens
 - Output tokens: \$4.40 per 1M tokens
- **Tool usage costs:**
 - Tool calls (e.g., code execution or browsing) incur separate fees in addition to token usage.

Metric	Agent A – Step 1	Agent X – Step 1	Agent B – Step 2	Agent Y – Step 2	Agent A – Step 3	Agent B – Step 3
Input Tokens	58567 ± 16066	45310 ± 13621	66557 ± 20801	52435 ± 12816	2724 ± 654	4681 ± 1906
Output Tokens	5555 ± 1156	6364 ± 1787	9829 ± 1993	7985 ± 2207	638 ± 496	1029 ± 979
Total Token Usage	64122 ± 16827	51675 ± 14470	76386 ± 21803	60420 ± 14200	3362 ± 469	5710 ± 2854
LLM Calls	11.08 ± 1.61	10.58 ± 1.93	13.89 ± 2.85	13.44 ± 2.41	9.00	7.00
Execution Time (s)	1.45 ± 0.77	2.36 ± 0.23	22.21 ± 7.11	22.66 ± 8.34	0.33	0.24
LLM Wait Time (s)	66.36 ± 17.59	73.08 ± 19.32	121.62 ± 20.67	98.34 ± 23.44	22.11	15.65

Table 3.7: Agent Communication Step-wise Token and Time Metrics for o4-mini

Metric	Value
Mean Completion Time (s) (Step 1)	143.25 ± 33.87
Mean Completion Time (s) (Step 2)	264.83 ± 32.62
Mean Completion Time (s) (Step 3)	253
Total Completion Time (s) (Step 1 – 3)	661.08
Success Rate (Step 1)	12%
Success Rate (Step 2)	9%
Success Rate (Step 3)	10%

Table 3.8: Time to Completion and Success Metrics Table

4 Conclusion

This project investigated a central question in the field of decentralized identity and autonomous agent design: Can large language model (LLM) agents function as autonomous digital entities, capable of proving their identity and capabilities without relying on a centralized authority? The results suggest that such a model is indeed feasible, enabled by the use of decentralized identifiers (DIDs) and verifiable credentials. While the prototype validates the foundational concept, several practical and ethical challenges remain.

Main contributions

- **Identity workflows that follow web standards.**
 - Agents using LangGraph and AutoGen can issue, show, and check Verifiable Credentials (VCs) using tool servers such as MCP and Langchain.
 - These agents can also share credentials between different domains using Google's A2A protocol. This is where the different LLM agents have shown they can work together using DID and VC standards.
- **A modular design that separates responsibilities.**
 - Core tasks like cryptographic operations, JSON-LD handling, and DID resolution are handled by dedicated services outside the LLM.
 - This separation made the system more reliable and easier to debug, especially when dealing with issues like rate limits or context-fetch failures.
- **Testing across different OpenAI models.**
 - GPT-4.1 worked best, with over 90% success on most tasks, though it had some trouble at the start of the process.
 - GPT-4o-mini was faster but failed often when issuing credentials.
 - Cheaper models like GPT-4.1-mini and O4-mini had lower success and took more time overall.
 - Overall, the tests show that **the quality of the model matters more than speed or price when it comes to completing the mutual authentication and VC issuance procedure correctly. However it also indicates that model performance can vary across different implementation frameworks.**

Open problems

1. **LLM unpredictability.** Sometimes the agents skip important steps or make decisions that don't follow the rules.
2. **Limitations of the A2A protocol.** The current flow is one-directional and role-fixed, limiting dynamic back-and-forth interactions between agents.

3. **Slow context fetching.** JSON-LD processing remains a bottleneck even with local caching. Embedding contexts or using offline bundles could improve reliability and speed.

What we learned

- *LLM agents benefit from explicit guidance:* Agents often hallucinate or skip steps unless the task is tightly scoped. Clear instructions and role definitions improve reliability.
- *Smarter is better than cheaper:* GPT-4.1 is worth the cost when you need things to work correctly.
- *Agents can follow identity protocols—but only just:* With enough prompting and structured tool use, LLMs can handle decentralized ID flows. But they remain brittle, and small changes in phrasing or order can cause failures.
- *Frameworks help enforce protocol discipline:* Tools like LangGraph and AutoGen introduce structure, memory, and coordination that make agents less prone to drift during multi-step identity exchanges.
- *Failure analysis is part of the design loop:* Many errors were non-obvious and required tracing full interactions. Logging, trace inspection, and retries were critical for debugging and improvement.
- *Cross Domain is possible but limited:* Google’s A2A protocol is still in its early stages and evolving quickly, thanks to open collaboration with the Linux Foundation and a wide range of technology partners. As a result, its features, best practices, and supported use cases are continually expanding. However, because agents cannot act as both client and server at the same time, real-world A2A systems typically need a centralized “Host Agent” or orchestration layer. This central component handles agent discovery, coordination, and communication. Overall, cross-framework agent communication using A2A worked, but the protocol’s one-way flow and fixed roles limited more flexible interactions.
- *Error tolerance must be built-in:* The system should support retries and fallbacks to handle failures like VP creation or DID resolution issues. Without this, small faults break the full authentication flow.
- *Resilience Against Prompt-Based Adversarial Manipulation:* The system should be built around the notion of securing the agent’s behavior by implementing a detection system for faulty interaction through rigorous prompt engineering. This can help mitigate adversarial influence (through the prompt) and agent hallucinations.

Next steps

1. **Make the system more robust,** adding better caching, retries, and protection against rate limits to keep it running 24/7.
2. **Continuously benchmark newly released LLMs** Evaluate their performance with regard to:
 - Compliance with roles and protocol steps
 - Speed vs. cost efficiency
 - Robustness against prompt drift and unexpected behavior

3. **Implement retry and validation logic** to handle failures like missing credentials, VP creation errors, or invalid responses. Add validation checkpoints after each interaction step.
4. **Introduce real-world use cases for evaluation**, e.g., AI agents for customs clearance or bank KYC verification, to benchmark performance under realistic conditions.
5. **Build agent health monitoring and recovery strategies** using watchdogs or timeout detection to identify infinite loops, failed authentication steps, or protocol violations and react accordingly.
6. **Future Potential for Symmetric Agents** can be re-evaluated in light of developments. Currently, A2A does not support scenarios where agents can freely switch roles or interact as true peers. For cases that require fully symmetric, peer-to-peer agent collaboration, the protocol still has room to grow. We look forward to future updates that may unlock richer and more flexible agent topologies, making truly decentralized, dynamic multi-agent systems possible.
7. **Offloading Identity Operations via Trusted Libraries** to enhance system efficiency, reliability, and maintainability. It is advisable to delegate core identity operations—such as Verifiable Credential issuance, validation, and mutual authentication—away from local agent logic or the MCP server. Instead, these processes should be handled by specialized, enterprise-grade libraries or external services specifically designed for robust identity management. By leveraging established tools for workflows like DID resolution, SD-JWT verification, and Verifiable Credential presentation exchange, organizations can significantly reduce implementation complexity and strengthen security and LLM performance.

Final thoughts

By combining modern AI agents with open digital identity standards, we've shown that it's possible to build **trustworthy, self-managed AI agents**. There's still work to do, especially around reliability, but we now have a solid foundation for building agents that not only say smart things but can also prove who they are.

List of Acronyms

AI	Artificial Intelligence
A2A	Agent-to-Agent
API	Application Programming Interface
CA	Credential Application
CORS	Cross-Origin Resource Sharing
DID	Decentralized Identifier
HTTP	HyperText Transfer Protocol
JWS	JSON Web Signature
JSON-LD	JavaScript Object Notation for Linked Data
LLM	Large Language Model
MCP	Model Context Protocol
PD	Presentation Definition
SDK	Software Development Kit
URI	Uniform Resource Identifier
VC	Verifiable Credential
VP	Verifiable Presentation
W3C	World Wide Web Consortium
KYC	Know Your Customer

Bibliography

- [1] Rao Surapaneni, Miku Jha, Michael Vakoc, and Todd Segal. A new era of agent interoperability, 2024. <https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/>.
- [2] W3C Credentials Community Group. Decentralized identifiers (dids) v1.0, 2023. <https://www.w3.org/TR/did-core/>.
- [3] W3C Credentials Community Group. Verifiable credentials data model v2.0, 2025. <https://www.w3.org/TR/vc-data-model/>.