ConverseAPI
Converse with any API

The Problem

Building LLM Agents from OpenAPI specifications in a CI/CD pipeline

In today's digital landscape, APIs are the backbone of countless applications, facilitating seamless data exchange and automation. However, the challenge of making these APIs truly interactive and user-friendly remains unresolved. More traditional API integrations via UIs often fail to provide a dynamic, conversational user experience, particularly when interacting with complex systems or enabling real-time decisions based on API data. As a result, those applications lack a practical, accessible usage experience while their APIs are not used to their full potential.

Developers face significant hurdles in bridging this gap. The process of transforming raw API responses into engaging, context-aware dialogues is complex, especially when dealing with a wide variety of user needs and behaviors. Additionally, the fast-paced nature of API changes adds pressure to rapidly develop and deploy conversational features without sacrificing quality or responsiveness.

ConverseAPI is a pioneering project designed to address these challenges by enabling APIs to "speak" directly to app users through advanced natural language interfaces. Leveraging the power of Large Language Models (LLMs), ConverseAPI enables developers to create API-driven conversational agents, that are not only responsive and context-aware but also adaptable to the specific needs of each application.

In this project we will focus on building a modular framework that integrates seamlessly with any application's OpenAPI specification, allowing for rapid development and deployment of end-user facing apps using conversational AI. With ConverseAPI, users are able to interact with very complex systems, such as financial platforms, IoT platforms, or content management systems, using intuitive natural language. Our solution will translate natural language user inputs to API call execution plans, generate the necessary data for all requests, and translates API responses into actionable, user-friendly dialogues.

But we will not stop at enhancing decision-making and user engagement: With ConverseAPI, developers will have all the tools they need to quickly integrate conversational AI in all of their apps and create sophisticated interfaces that bring their existing APIs to life. ConverseAPI supports developers in all development phases, from integration in existing apps over interactive API exploration and debugging to automatically generating API test cases and monitoring usage.

Together we engineer how APIs will be used in the future!

Scenarios

Scenario 1: Natural Language API usage and Explainability via APIs

The app **IQvolt** provides its users with real-time information about their photovoltaic system, home battery charging, and registered IoT smart home appliances. Furthermore, it is able to schedule energy intake from grid electricity, energy storage and execution of energy demanding smart home appliances based on historic data analysis and forecasts. IQvolt aims for maximizing self-produced or cheaply available energy usage while being as transparent in its decisions to its users as possible. Therefore, IQvolt uses a ConverseAPI integration allowing natural language API interactions.

IQvolt app user Chaima notices in her IQvolt app dashboard that her home battery storage is currently charged by the grid, even though energy prices are currently comparably high. She does not understand why her IQvolt system decided to do so and asks her watchOS app in natural language: “Why is the battery currently being charged? The prices seem higher than usual.”. As a result, her watchOS app forwards her request to ConverseAPI, which in turn creates a sequence of API-calls to IQvolt’s server, fetching all information needed to answer Chaima’s question.

As soon as ConverseAPI receives all needed responses, it formulates a natural language answer, which is forwarded to Chaima’s IQvolt watchOS app: “Good catch! Your battery was almost empty and the energy cost is forecasted to be two times higher within in the next few hours, whilst a significant increase in consumption is anticipated at the device: “Stove” and “Oven”. Because of this, I decided to charge your home battery storage to 80%.”. Indeed, Chaima planned to start cooking dinner within the next few hours, so she says: “Thank you! But for tomorrow, you do not need to plan this: I will be eating at a restaurant.”. Again IQvolt produces an answer using ConverseAPI’s IQvolt integration and calling IQvolt’s user presence API. As an answer ConverseAPI assures Chaima, that it will take her preferences into respect for tomorrow.

Scenario 2: Easy Integration for Developers

Lara is a Weptun developer and wants to provide a natural language user interface to her already existing event management app **Beyond.Host**, in which event location guests can call for service staff, create on-the-fly votings, and control the event location’s media system. To realize this, Lara gathers all needed information to configure her own, cloud hosted, ConverseAPI integration:

- A link to Lara’s OpenAPI specification for her Beyond.Host app
- Credentials for accessing Beyond.Host’s API
- Credentials for her favorite on-premises LLM server

With all information gathered, Lara opens her MacBook, starts the ConverseAPI app, and signs into her ConverseAPI account. On the starting page, she clicks “Add new integration” and fills all fields with the data she prepared. After Lara’s confirmation, ConverseAPI creates a new integration for her app and displays a new dashboard for Beyond.Host. From here Lara can see all statistics about her new integration’s usage, previously generated API execution plans, errors when calling Beyond.Host’s API, and a link to a ConverseAPI’s endpoint, to which Lara can send streaming natural language requests.

Lara adds a chat window to Beyond.Host and calls the ConverseAPI endpoint with user entered natural language data. To test her new integration, Lara writes “Can you please turn on the heating in my room and call someone to bring some refreshments?”. ConverseAPI dynamically generates a series of API requests to Beyond.Host: Get room information, fetch attached IoT heating devices, increase temperature, check service staff availability, and call staff. From ConverseAPI’s Beyond.Host integration dashboard, Lara verifies all generated LLM prompts, context information, generated API call execution plans and used data transfer objects of her request. Everything seems alright to her on the dashboard and she is very happy, as she sees, that the heater changes and her test device notifies her about an open service call.

Scenario 3: Interactive API Exploration and Dynamically Generated UI

Robert is a freelancer and supports Weptun at developing **Lehrer-App**, a very large software system aiming for supporting teachers in their daily tasks. As a freelancer Robert is required to quickly familiarize himself with existing code and a system's APIs in order to start writing his own code as fast as possible. Soon in the project Robert realizes, that Lehrer-App's API is extensive and utilizes detailed and deeply nested data transfer objects (DTOs). Luckily, Weptun has already integrated ConverseAPI in Lehrer-App, so Robert decides to start an interactive exploration of its complex API in order to start developing a new Lehrer-App feature.

Robert begins a chat with ConverseAPI in Lehrer-App's integration dashboard: "How can I create a new school class in the system?". ConverseAPI processes this request and constructs an API execution plan, which it prints in natural language: "First you need to create a school using the `POST /school` endpoint, since `SchoolClass` DTOs are required to be assigned to a school. After that, you create a new `SchoolYear` by calling `POST /schoolyear`, referencing the created school's ID from the first endpoint's response in the DTO's `schoolId` body field. Classes are always unique for a given school and school year combination. As a last step, you create a `SchoolClass` via a `POST request to /schoolclass`. In this request you need to reference both, the created school's ID in `schoolId` and the school year's ID in `yearId`. Also, you might want to add some students to the created class via the `POST /student` endpoint, since many endpoints require class assigned `Student` DTOs as an input. Here is an overview of my suggested request execution plan:

1. Call `POST /school` using a `School` DTO
2. Call `POST /schoolyear` using a `SchoolYear` DTO, referring to the school's ID from step 1′s id response in the `schoolId` field
3. Call `POST /schoolclass` using a `SchoolClass` DTO, referring to the school's ID from step 1 in `schoolId` and using the school year's ID from step 2 in the `SchoolClass` DTO's `yearID` field"

Robert is now familiar with this part of Lehrer-App's API, so he starts programming the new feature right away.

As Robert finishes writing code, he wants to manually test his day's work. To do so, Robert requires a `SchoolClass` object in Lehrer-App's database. From his past ConverseAPI interaction, he knows that he needs at least three different multi-level DTOs to create a `SchoolClass`. As creating meaningful and detailed sample data is quite time-consuming, he asks ConverseAPI for assistance: "Can you provide me with DTOs to create a new school class?". "Sure. Do you want RAW data or an interactive UI for testing the API with sample data?", ConverseAPI responds. Robert types: "I would like to have an interactive UI."

After Robert's confirmation, ConverseAPI generates context sensitive data for the first request to create a new school, by using the value "Gymnasium Friedrichshafen" as sample input for the school's name field. In order to generate a dynamic and interactive UI, ConverseAPI infers all data type information from Lehrer-App's OpenAPI document and presents text boxes, drop-downs, and list UI elements to fill each field. As a last step, ConverseAPI populates all generated input elements and annotates each component with the expected field's contents and context hints like "Required" or "Previous step's school ID" whenever applicable. Robert can now edit all pre-populated values as he pleases. At the bottom of the form, Robert finds a button to execute the request with his provided data and proceed to the next request of the execution plan. In an adjacent text-box Robert can always see Lehrer-App's previous responses to his API requests.

Scenario 4: Natural Language API Tests

Mark works in quality assurance at Weptun and wants to create integration tests for the brand new **Syncnet** app, that acts as a middleware for several software systems, providing a unified REST-API to receive data and broadcast it via WebSockets to other systems. Since time and budget of Mark's customer is limited, Mark wants to automatically generate some basic test cases for his Java JUnit test suite.

To realize this, Mark adds a Syncnet integration to ConverseAPI and writes tests for Syncnet's API in natural language:

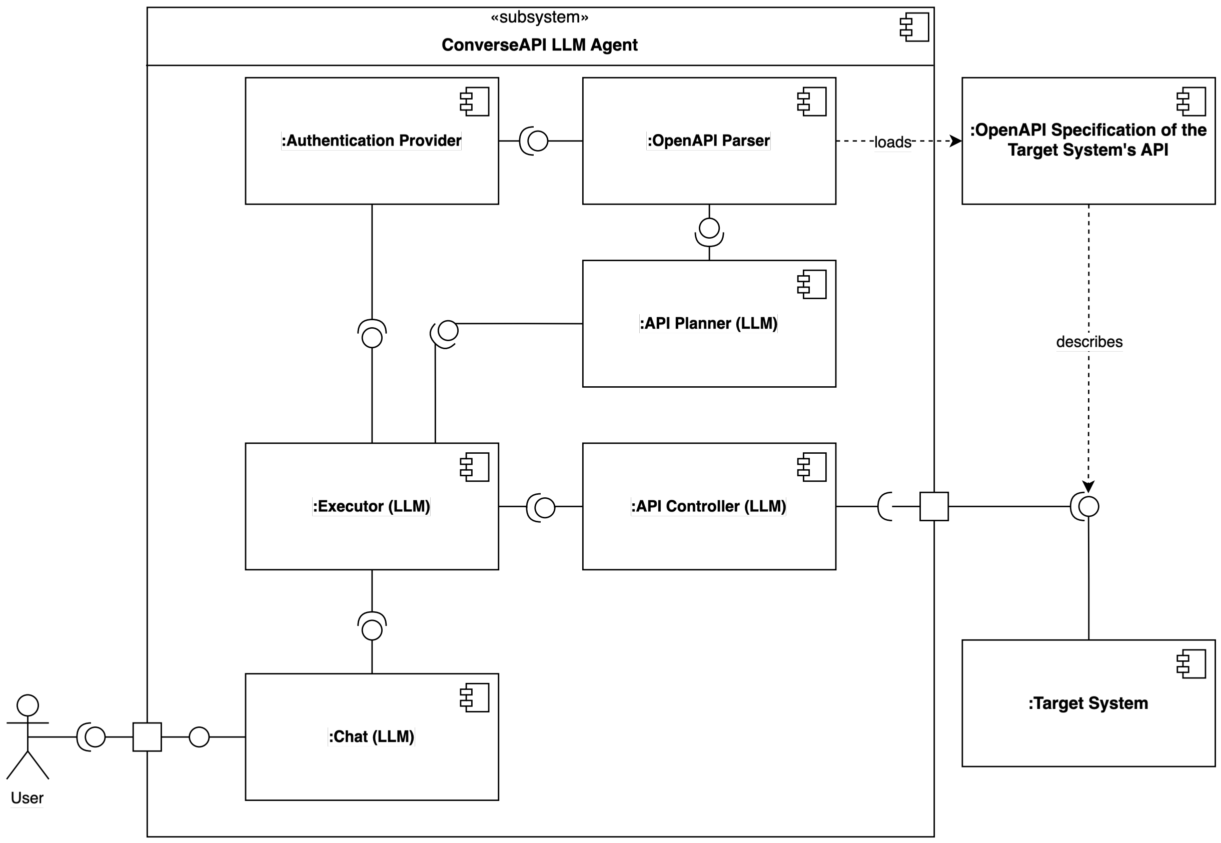
1. Whenever Syncnet receives data at the `/metering-data` endpoint, all other systems should be notified about this new data point.
2. Whenever the `DELETE /metering-locations/{id}` endpoint is called, there should be a "Forbidden" response if the requesting user doesn't have admin privileges.
3. All endpoints should return a `2xx` status if all required fields are filled with meaningful data.
4. All endpoints should validate their inputs according to the OpenAPI specification.

From this description, ConverseAPI automatically generates Java code to test Syncnet's API:

1. Subscribe to the `/metering-data` WebSocket => Post sample data to `/metering-data` => Test if the sample data was received via the WebSocket.
2. Call `DELETE /metering-locations/{id}` with non-admin user and validate, that the response code is `403`, call the same endpoint again using an admin user and assert a successful response.
3. Generate DTOs for all endpoints using minimal required data and send requests expecting successful responses.
4. Generate invalid DTOs, e.g. having too few characters, list items or wrong ID data and assert `400` answers.

Mark is now able to copy the generated test functions to his existing test suite with minimal effort and integrates them in Syncnet's continuous development pipeline.

Top-Level Design



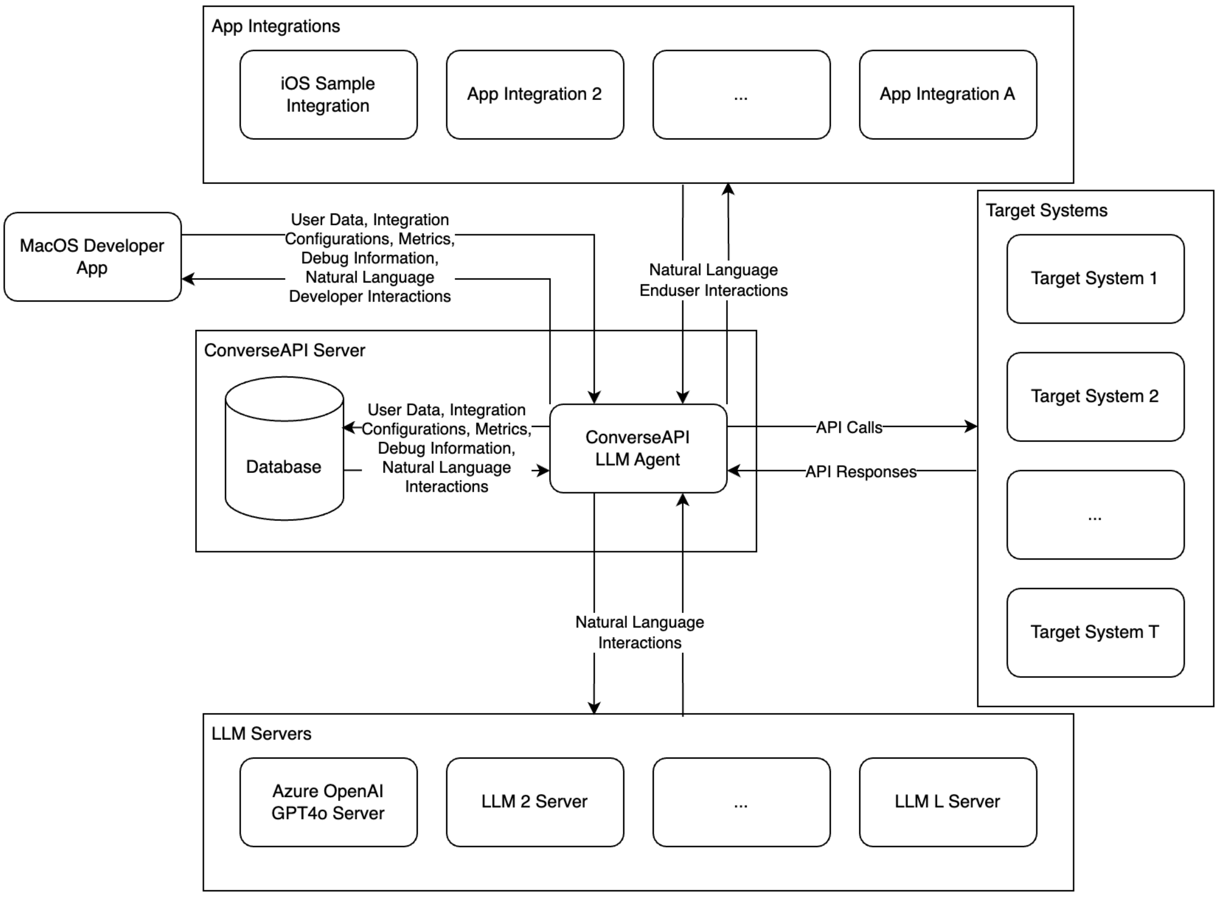
Our current vision of ConverseAPI's Top-Level Design is a multi-level composite LLM Agent architecture, which in turn results in an LLM Agent on its own: The **ConverseAPI LLM Agent**. In this iPraktikum you will refine this subject-to-change design, implement it, provide UI components for natural language interaction, monitoring its state, and debugging potential problems. All components presented here are just suggestions by Weptun. Our architecture is heavily inspired by the Hierarchical Planning Agent example from LangSmith[1] in order to limit context information size for each LLM request.

In the past years, Weptun developed several **Target Systems**, which are well-documented using OpenAPI specifications. Those systems can be added as an integration to ConverseAPI. Each integration instance defines its own ConverseAPI LLM Agent, to which **User**'s natural language inputs are sent. Internally, this agent maintains a **Chat** LLM-Agent to interact with the User as already established by existing LLM agents, such as ChatGPT or Llama. As soon as the Chat Agent detects, that an API call is needed to fulfill a User's request, it calls the **Executor** LLM-Agent with all relevant context information. This agent is responsible for holding the input context and requesting an API request execution plan from the **API-Planner** LLM-Agent, which is aware of the target system's parsed (**OpenAPI Parser**) **OpenAPI Specification** and required authentication (**Authentication Provider**) information, which can be requested from the user via the Executor and Chat-Agents if needed. As soon as the requests, required headers and payloads are defined, the Executor fills all payload fields with its held contextual information, adhering to eventual API limitations, such as required fields, expected list lengths or field content validations. As soon as all requests are defined, the final execution plan and relevant context is sent to the **API Controller** LLM-Agent. This subsystem monitors request

[1]https://python.langchain.com/docs/integrations/tools/openapi/#1st-example-hierarchical-planning-agent

execution and implements smart reactions to API responses, such as “Bad Request"s by modifying the request if it can resolve the error e.g. using the target system’s error response detail information. As soon as all required requests finished, information is back propagated to the Chat Agent via the Executor Agent.

Initial Architecture


App Integrations
iOS Sample Integration
App Integration 2
...
App Integration A
MacOS Developer App
User Data, Integration Configurations, Metrics, Debug Information, Natural Language Developer Interactions
Natural Language Enduser Interactions
Target Systems
Target System 1
Target System 2
...
Target System T
ConverseAPI Server
Database
User Data, Integration Configurations, Metrics, Debug Information, Natural Language Interactions
ConverseAPI LLM Agent
API Calls
API Responses
Natural Language Interactions
LLM Servers
Azure OpenAI GPT4o Server
LLM 2 Server
...
LLM L Server

We propose the following architecture as a starting point for ConverseAPI. The system consists of a **server**, running **ConverseAPI**’s services, such as its natural language API and a RESTful-Interface to fetch all relevant data for the **Developer App**. An arbitrary number of **App Integrations** sends their user’s natural language messages to **ConverseAPI’s LLM Agent**, which inturn calculates responses and forwards necessary API calls to integrated **Target Systems**. One App Integration corresponds to one or many Target Systems. All natural language requests of developers and end users are forwarded to a configurable **LLM-Server**. One LLM-Server corresponds to one or many App Integration. All calculated responses are sent form the CovnerseAPI LLM Agent bundled with Target System API responses in natural language back to the requesting system. To measure ConverseAPI’s success and provide developers a means to debug the system, ConverseAPI stores all interactions and other metrics in a **Database** for future retrieval.

Requirements and Constraints

Functional Requirements

FR1: OpenAPI Target System Integration (High Priority)
The system should be able to interact with any well-documented OpenAPI using natural language input, OpenAPI documentation authentication and target system API authentication.

FR2: Debugging UI for Developers (High Priority)
The system should provide a user interface that helps developers understand underlying LLM and API calls. Specifically it displays inputs, LLM context, generated execution plans, individual endpoint calls with headers and payloads, accuracy metrics, latency, and the number of used tokens.

FR3: API Exploration UI (High Priority)
Developers should be able to explore APIs in natural language. The system answers questions about the API and relevant. Furthermore, the system is able to display API request execution plans, provide reasoning on why those requests need to be made, and explains how different API-calls are related to each other.

FR4: Generated Test Cases (High Priority)
The system should be able to generate API Test Cases from natural language descriptions in a developer's preferred programming language.

FR5: User Confirmation Before Committing to Actions (High Priority)
The system should require user confirmation before executing actions with irreversible consequences, such as sending mails or deleting data.

FR6: Monitoring Dashboard for Analyzing Usage and Error Rates (Medium Priority)
The system should provide a monitoring dashboard to analyze usage, error rates, and other performance metrics, allowing administrators to track error rates and monitor usage patterns.

FR7: Automatic API Change Detection and Notification (Medium Priority)
The system should automatically detect changes in the API definition, adapt its internal LLM functions and notify developer(s) about the detected changes. It should furthermore describe the changes in natural language, which can be sent to third party API users.

FR8: Dynamic User UI for Parameter Filling (Medium Priority)
The user interface should dynamically generate input components to fill DTO fields relevant for any given request execution plan and explain an individual fields expected input and its semantics.

FR9: Custom User UI for Parameter Filling (Medium Priority)
The user interface should be able to display per-integration developer pre-defined UI components to fill DTO fields relevant for any given request execution plan and explain an individual fields expected input and its semantics, such as color-pickers or styled URL components.

FR10: Automatic Detection of Unexpected API Behaviour (Low Priority)
The system should automatically detect unexpected behavior of API endpoints, e.g., missing specification of a required field in API documentation. Furthermore, it should notify the developer, but the developer should be able to “ignore” to guard against false positives.

FR11: Integration Framework (Low Priority)
Developers should have the option to integrate a framework in their app to display dynamically generated UI components tailored to their app’s functionality and to write their own LLM functions.

Non-Functional Requirements

NFR1: Interchangeable LLMs (High Priority)
The system should provide its functionality by utilizing LLMs, which can be selected individually for each integration.

NFR2: Easy Initial Configuration (High Priority)
The system should allow easy creation of integrations and their configurations, such as setting the URL of the OpenAPI specification.

NFR3: Support of Large API Specifications (Medium Priority)
The system should support large API specifications (e.g., with more than 30 endpoints) by automatically clustering in sub-agents.

NFR4: Answer User Requests in Reasonable Time (Medium Priority)
The system should generate responses to users within 10 seconds.

Constraints

The following technologies should be used:
- Cloud-hosted LLMs (e.g. Azure OpenAI GPT-4o)
- Self-hosted LLMs (e.g. smaller models like Llama 3 8B)
- SwiftUI for MacOS and iOS apps

The following technologies are recommended:
- LangChain: Framework to build LLM agents
- Docker for easy deployment
- Relational database using PostgreSQL

Development

The project will target iOS 18 and macOS 15. Weptun and TUM will provide the necessary infrastructure, including access to the latest models on Azure OpenAI, such as GPT-4o.

Environment

- Cloud server VM for hosting the LLM Agents
- Cloud server VM for hosting the server-side services
- iPhones using the latest iOS version
- MacBooks using the latest MacOS version
- OpenAPI definitions used in production

Deployment
At the end of the iPraktikum, ConverseAPI will be integrated in many projects of Weptun and therefore, is confronted with a very heterogeneous set of API functionalities.

Client Acceptance Criteria

This is a broad problem statement, so the requirements stated above might change. The customer is willing to discuss and change the requirements. The project team should handle changes in an agile way. The work items are prioritized with the client during the project in Sprints.

Deliverables

The project's deliverables are:

- MacOS App: Desktop app for developers to manage the complete life-cycle of the API integration
- iOS App: Proof-of-concept mobile app, representing a ConverseAPI integration for end user interaction
- Server: Server application(s) as a Docker container
- Build scripts and installation instructions
- Documentation: System Design Document (SDD), Administrator Manual, and a Requirements Analysis Document (RAD), which includes:
  - Object Model
  - Functional Model

Schedule

The project consists of the following milestones:

- Kickoff Meeting: Thu, 17 October 2024, 5:00 p.m.
- Design Review: Thu, 12 December 2024, 5:00 p.m.
- Client Acceptance Test: Thu, 6 February 2025, 5:00 p.m.

Weekly SCRUM meetings with the project management.

Meetings with the customer every two weeks.