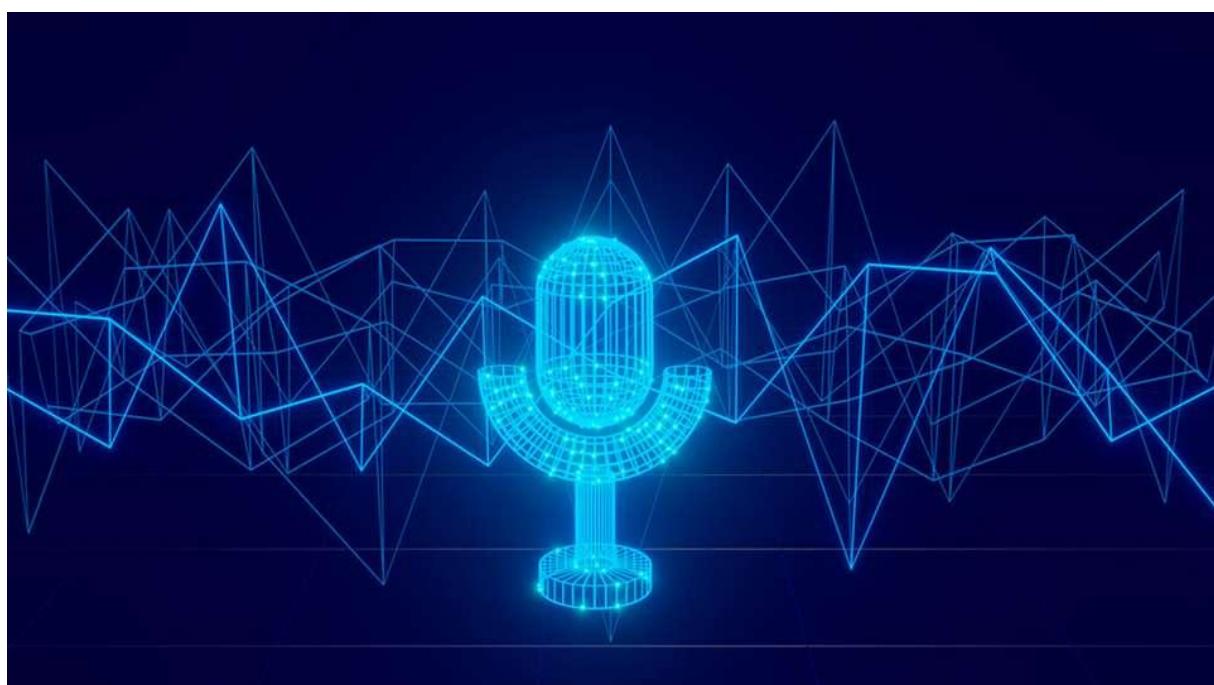


Mechatronisches-Projekt Spracherkennung



Ronquillo-Luna, Alan

Tan, Levi

Geng, Xintong

Inhalt

Introduction	1
Project Overview	1
Motivation and Objectives	1
System Architecture	1
Key Features	2
Sprache	3
Voraussetzungen	3
Wozu ngrok?.....	4
Ngrok installieren und einrichten.....	4
N8n Container konfigurieren.....	4
Whisper installieren und einrichten.....	5
Telegram Account anlegen und Credential in n8n anlegen	6
Aufbau in n8n	6
Sprachverarbeitung im Workflow	7
Warum Telegram für Sprache?	8
AI Agent Operation in n8n and Local Ollama Deployment.....	9
AI Agent Functionality in n8n	9
Role of Ollama in the AI Agent Architecture	9
Installation and Startup of Ollama and n8n	10
Summary	10
Python Automation Engine ('Spracherkennung.py').....	11
Prerequisites & Setup.....	11
Layer 1: The Dispatcher (Routing Logic).....	11
Layer 2: The Driver (Execution Logic)	12
Localization: Bilingual Interface Support.....	12
How to Run	13
Summary of Architecture	13
Installation auf Linux-Rechner	14
Fazit.....	14

Introduction

Project Overview

This project focuses on the development of an intelligent, voice-controlled automation system for the SAP Fiori environment, specifically targeting the "Advanced Manufacturing" module. The primary objective is to enable users to control complex ERP processes—such as logging in, navigating to specific orders, and managing production statuses—using natural language voice commands via Telegram.

By integrating Generative AI (Ollama) with Robotic Process Automation (Selenium), this system bridges the gap between unstructured human speech and rigid system interfaces, creating a hands-free solution for production environments.

Motivation and Objectives

Manual interaction with SAP systems often involves repetitive tasks and complex navigation paths. The goal of this project was to automate these routine procedures to increase efficiency and reduce the cognitive load on operators.

Key objectives defined for this project include:

- Voice-to-Action: Establishing a reliable channel for voice input using Telegram and Whisper (Speech-to-Text) since no specific input channel was mandated
- Modularity: Creating a robust framework that allows future project groups to easily adopt and extend the system with new use cases
- Robustness: Ensuring the system can handle UI latency and language variations (English/German) without failure

System Architecture

To ensure scalability and maintainability, the system follows a 3-Layer Architecture as defined in the project milestones:

- Intent Layer (AI Agent & n8n):

This layer handles the ingestion of voice messages via Telegram and transcribes them using Whisper running in a Docker container.

n8n orchestrates the workflow, sending the text to a local Ollama LLM. The AI Agent interprets the user's intent and converts it into a standardized JSON command (e.g., {"action": "start_order", "params": {...}}).

ngrok is used to tunnel these requests securely to the local environment.

- Dispatcher Layer (Flask Server):

Implemented in Python, this layer acts as the controller. It receives the JSON payload from n8n via an HTTP endpoint, validates the requested action, and routes it to the specific robotic function.

- Action Layer (Selenium Robot):

The execution core, powered by Selenium WebDriver. It interacts directly with the SAP Fiori DOM to perform physical clicks and text inputs.

This layer includes advanced error-handling mechanisms, such as "auto-fix" logic for filters and a "triple-click" strategy to ensure actions are executed reliably.

Key Features

- Bilingual Interface Support: The automation engine automatically detects and adapts to both English and German SAP interfaces, dynamically mapping keywords (e.g., "Start" vs. "Starten")
- Containerized Deployment: Core components like n8n and Whisper are deployed via Docker to ensure a consistent development environment across different machines
- Dynamic Application Handling: The system can intelligently identify and open SAP applications even if the spoken name differs slightly from the official tile name (e.g., mapping "Capture of production data" to "Erfassung von Produktionsdaten")

Sprache

In diesem Projekt wurde eine Sprachsteuerung für ein Produktionsumfeld mit n8n, Whisper (Speech-to-Text) und einem AI-Agent umgesetzt. Da in der Aufgabenstellung kein fester Kanal für die Spracheingabe vorgegeben war, musste zunächst ein geeigneter Weg gefunden werden, wie Sprachbefehle zuverlässig ins System gelangen. Als praktikable Lösung wurde Telegram gewählt, weil Sprachnachrichten dort einfach aufgenommen und über Webhooks direkt an n8n weitergeleitet werden können. Damit diese Webhooks auch bei lokal laufendem n8n funktionieren, wird in der Entwicklungsumgebung ngrok verwendet, um eine öffentliche HTTPS-Adresse bereitzustellen. Die folgende Dokumentation beschreibt die Einrichtung Schritt für Schritt, sodass der Aufbau auch von anderen Nutzern reproduzierbar nachgebaut werden kann.

Voraussetzungen

- **Docker**

- Docker Desktop ist installiert und läuft
- Docker Version 28.5.1
- Docker Compose Version v2.40.3-desktop.1

- **n8n**

- n8n läuft lokal als Container
- n8n ist im Browser erreichbar unter <http://localhost:5678>
- n8n Version 1.121.3

- **Telegram**

- Ein Telegram Bot ist über BotFather erstellt
- Ein gültiger Bot Token liegt vor
- Ein Test Chat für Sprachnachrichten ist vorhanden

- **ngrok**

- ngrok ist installiert
- Authtoken ist eingerichtet
- Ein aktiver HTTPS Tunnel leitet auf den lokalen n8n Port 5678 weiter
- Die ngrok HTTPS URL ist in n8n als Editor Base URL und Webhook URL eingetragen

- **Whisper**

- Whisper läuft als HTTP ASR Service in Docker
- Image onerahmet openai-whisper-asr-webservice latest
- Aus n8n erreichbar über <http://whisper:9000/asr> im Docker Netzwerk
- Port 9000 ist frei

Wozu ngrok?

Für Telegram und viele andere Dienste müssen Webhooks über eine öffentlich erreichbare HTTPS URL aufgerufen werden können. Da n8n lokal meist nur unter `http://localhost:5678` läuft und von außen nicht direkt erreichbar ist, löst ngrok dieses Problem, indem es einen sicheren Tunnel von einer öffentlichen Adresse auf den lokalen n8n Port erstellt. Diese URL wird anschließend in n8n als Basis URL beziehungsweise Webhook URL eingetragen, sodass n8n beim Erzeugen von Webhooks nicht mehr localhost, sondern die ngrok Adresse verwendet.

Ngrok installieren und einrichten

Man installiert ngrok (<https://dashboard.ngrok.com/get-started/setup/windows>), indem man das passende Paket für ein Betriebssystem herunterlädt und entpackt. Anschließend verknüpft man ngrok einmal mit einem Account über den Authtoken, den man im ngrok-Dashboard erhält. Typischerweise ist es dann ein Befehl wie `ngrok config add-authtoken <DEIN_TOKEN>`.

Anschließend startet man den Tunnel auf den n8n-Port 5678, beispielsweise mit dem Befehl `ngrok http 5678`. Ngrok zeigt daraufhin eine Forwarding-Adresse an, beispielsweise `https://treefrog-enabling-leopard.ngrok-free.app`. Genau diese HTTPS-URL benötigt man später für die n8n-Konfiguration.

N8n Container konfigurieren

In Docker Desktop wird der n8n-Container mit folgenden Kerndaten erstellt

- **Container Name:** z. B. n8n-container-ngrok
- **Ports:** Host 5678 → Container 5678/tcp
- **Volume:** lokaler Host-Ordner (eigener Pfad) → Containerpfad /home/node/.n8n

Dadurch bleiben Workflows und Credentials persistent, auch wenn der Container neu gestartet wird.

Anschließend werden unter Environment Variables (entscheidend für ngrok/HTTPS) diese Werte gesetzt

- `N8N_COMMUNITY_PACKAGES_ALLOW_TOOL_USAGE = true`
- `N8N_EDITOR_BASE_URL = https://DEINE_NGROK_URL`
- `WEBHOOK_URL = https://DEINE_NGROK_URL`
- `N8N_DEFAULT_BINARY_DATA_MODE = filesystem`

Wichtig ist dabei: Für N8N_EDITOR_BASE_URL und WEBHOOK_URL muss exakt die HTTPS-Forwarding-URL aus ngrok verwendet werden. Danach wird der Container gestartet. Falls ngrok später eine neue URL erzeugt, müssen diese beiden Variablen entsprechend aktualisiert werden (oder es wird eine feste ngrok-URL verwendet).

Whisper installieren und einrichten

Für die Spracherkennung wird Whisper nicht lokal „klassisch“ installiert, sondern als Docker-Container betrieben. Zunächst wird geprüft, ob Docker (inklusive Docker Desktop) installiert und ausgeführt wird. Anschließend wird das Whisper-Image aus der Registry geladen und als Container gestartet. Der Container muss auf Port 9000 laufen, damit der ASR-Webservice später über HTTP erreichbar ist.

Wichtig ist, dass der Whisper-Container im gleichen Docker-Netzwerk wie n8n läuft. So kann n8n den Dienst nicht über localhost, sondern über den Container-Namen erreichen, beispielsweise `http://whisper:9000/...` Deshalb wird der Container mit einem festen Namen wie „whisper“ gestartet und in das Netzwerk von n8n gelegt. Zusätzlich wird ein Volume für den Cache gesetzt, damit Modelle und Zwischendaten nicht bei jedem Neustart neu geladen werden müssen.

Beispiel (Terminal)

```
bash

docker network create n8n-net 2>/dev/null || true

docker run -d --name whisper \
--restart unless-stopped \
--network n8n-net \
-p 9000:9000 \
-e ASR_MODEL=base \
-e ASR_ENGINE=openai_whisper \
-v whisper-cache:/root/.cache \
onerahmet/openai-whisper-asr-webservice:latest
```

Nach dem Start wird geprüft, ob der Container läuft (z. B. in Docker Desktop oder mit dem Befehl `docker ps`). Danach kann n8n den Dienst über den HTTP-Request-Node ansprechen. Als URL wird intern (innerhalb des Docker-Netzwerks) der Containername verwendet, beispielsweise:

`http://whisper:9000/asr?output=json&task=transcribe&language=de&encode=true`

Damit ist der Whisper-Dienst installiert und so eingerichtet, dass n8n Audiodateien an den Container senden und die Transkription als JSON weiterverarbeiten kann.

Telegram Account anlegen und Credential in n8n anlegen

Bot + Token erstellen (Telegram)

- In Telegram den Chat @BotFather öffnen
- Befehl /newbot ausführen, Name + Username vergeben
- BotFather stellt den HTTP API Token bereit. Dieser Token wird anschließend in n8n als Zugangsdaten für die Telegram-Anbindung benötigt

Credential in n8n anlegen

- In n8n oben rechts oder links im Menü: Credentials → New
- Credential-Typ: Telegram API (oder "Telegram")
- Feld Access Token = den Token von BotFather einfügen
- Speichern, z. B. als Telegram_Bot

Aufbau in n8n

1) Telegram Trigger

- **Node hinzufügen:** Telegram Trigger
- **Credentials:** Telegram Bot Token (Telegram API Credential)
- **Updates:** Message

2) Get a file

- **Node hinzufügen:** Telegram („Get a file“)
- **Resource/Operation:** Get a file / Download
- **Credentials:** derselbe Telegram Credential
- **File ID (Expression):** {{\$node["Telegram Trigger"].json.message.voice.file_id}}

3) HTTP Request1

- **Method:** POST
- **URL:**
(<http://whisper:9000/asr?output=json&task=transcribe&language=de&encode=true>)
- **Authentication:** None
- **Send Body:** ON
- **Body Content Type:** Form-Data
- **Body Parameters:**
 - **Parameter Type:** n8n Binary File
 - **Name:** audio_file
 - **Input Data Field Name:** data

4) Code in JavaScript (Text sauber rausziehen)

- **Node hinzufügen:** Code (JavaScript)
- **Ziel:** aus der Whisper-Antwort einen klaren Text machen und als Feld speichern.

5) Edit Fields (Manual Mapping)

- **sessionId** (String): {{\$node["Telegram Trigger"].json.message.chat.id}}
- **action** (String): wird fest auf **sendMessage** gesetzt
- **chatInput** (String): {{\$json.cleanText}}

6) AI Agent (LLM Processing)

- **Node hinzufügen:** AI Agent
- **Prompt (User Message):** {{ \$json.chatInput }}
- System Message:** „Du bist ein Automatisierungs- und Mechatronik-Assistent...“
- **Verbundene Sub-Nodes:**
 - Model:** Ollama Chat Model (Llama3 via Localhost).
 - Memory:** Simple Memory (Window Buffer) für den Kontext.

7) HTTP Request (Send to Python Backend)

- **Node hinzufügen:** HTTP Request
- **Method:** POST
 - URL: http://host.docker.internal:5000/execute
- **Body:** {{ \$json["output"] }}

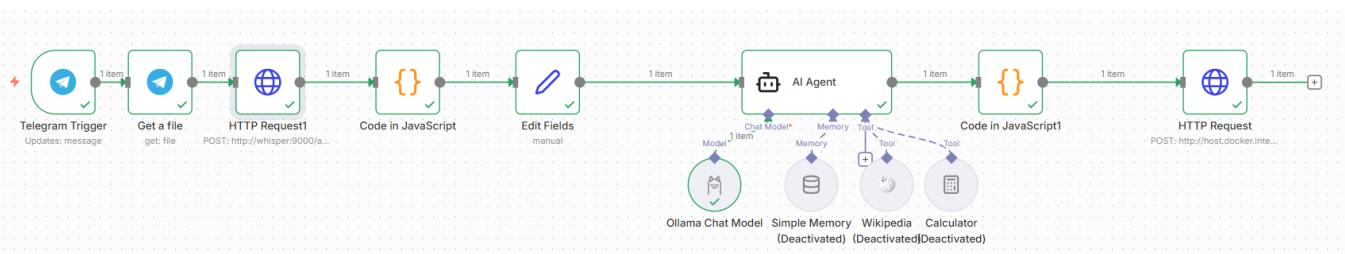


Abbildung 1: n8n-Aufbau

Sprachverarbeitung im Workflow

Der Workflow wird über den Telegram-Trigger gestartet. Sobald eine neue Sprachnachricht eingeht, übernimmt n8n die Daten und startet die weitere Verarbeitung.

Im nächsten Schritt wird die Audiodatei aus Telegram geladen und an den Whisper-Dienst weitergegeben. Dieser wandelt die Sprache in Text um, sodass anschließend nicht mehr mit Audio, sondern mit einem normalen Textstring gearbeitet werden kann.

Um diesen Text später besser verstehen zu können, folgt ein Code Node zur Textbereinigung.

Dabei wird der Text zuerst vereinheitlicht, also kleingeschrieben. Leerzeichen werden korrigiert und Satzzeichen am Ende entfernt. Anschließend werden typische Fehlhörer ersetzt, zum Beispiel falsche Varianten von „öffne“, „SAP“ oder „Advanced Manufacturing“. Zusätzlich werden verschiedene Schreibweisen bestimmter App-Namen vereinheitlicht, damit am Ende möglichst immer dieselben Begriffe im Text stehen.

Anschließend wird der bereinigte Text auf bekannte Kommandos geprüft. Beginnt der Text mit „öffne“, wird versucht, den App-Namen auch dann korrekt zuzuordnen, wenn er leicht falsch erkannt wurde. Dazu wird der restliche Text mit einer Liste erlaubter Apps verglichen, um die passende App zu ermitteln. Für alle anderen Kommandos wird der Text ebenfalls mit einer festen Liste typischer Befehle verglichen, sodass am Ende ein klarer Standardbefehl entsteht, zum Beispiel „Öffne SAP“ oder „Nächster Auftrag“. Dadurch wird verhindert, dass kleine Sprach- oder Schreibfehler den Workflow komplett durcheinanderbringen.

Das standardisierte Ergebnis wird anschließend an den AI Agent übergeben.

Dieser erstellt daraus ein eindeutiges JSON-Kommando, das später vom Selenium Robot ausgeführt werden kann.

Im Node Edit Files wird die Ausgabe schließlich nochmals strukturiert, sodass die nachfolgenden Nodes immer dieselben Felder und ein konsistentes Datenformat erhalten. Dadurch bleibt der Workflow stabil und die Übergabe zwischen den einzelnen Schritten funktioniert zuverlässig.

Warum Telegram für Sprache?

Wir haben uns für diesen Weg entschieden, da er ohne eigene App-Entwicklung auskommt und dennoch im Produktionskontext zuverlässig funktioniert. Telegram dient dabei als einfache Oberfläche für Sprachbefehle, Whisper übernimmt die Transkription von Sprache zu Text und n8n verbindet alle Schritte zu einem klaren, modularen Workflow, der sich leicht erweitern lässt. Docker stellt sicher, dass die benötigten Komponenten reproduzierbar laufen und bei anderen Nutzern schnell genauso aufgesetzt werden können, ohne das System mit vielen lokalen Installationen zu belasten. Zusätzlich mussten wir ngrok einsetzen, da Telegram Webhooks nur über eine öffentlich erreichbare HTTPS-Adresse akzeptiert und n8n bei uns sonst nur lokal unter <http://localhost:5678> erreichbar war. Dadurch reagierte Telegram wiederholt mit „Bad Request“. Durch den ngrok-Tunnel steht eine gültige HTTPS-URL zur Verfügung, die auf den lokalen n8n-Port weiterleitet. Dadurch kann Telegram den Webhook korrekt registrieren und die Sprachnachrichten erreichen zuverlässig den Workflow.

AI Agent Operation in n8n and Local Ollama Deployment

AI Agent Functionality in n8n

The AI Agent in this project works as a logic and decision layer inside n8n. Its main task is to convert processed natural language input into one clear and executable JSON command that can be used by the automation scripts.

Once the voice input has already been transcribed and cleaned earlier in the workflow, the final text is sent to the AI Agent node. This agent uses a local language model running on Ollama and follows a strict prompt that defines exactly how the output must look. Because of this, the model always returns only one JSON object, without extra text or explanations.

The AI Agent is responsible for:

- Understanding the user's intent from the text
- Mapping the intent to a predefined action
- Extracting the required parameters from the command
- Applying default values when some information is missing
- Generating a standardized JSON output that can be executed safely

This approach reduces ambiguity and makes sure that small language differences do not break the automation workflow. n8n can therefore reliably connect human input with automated system actions.

Role of Ollama in the AI Agent Architecture

Ollama is used as the local engine that runs the AI model for the n8n AI Agent. Instead of using cloud-based services, the model runs locally inside a Docker container. This gives full control over the model, the prompt, and the data flow.

n8n communicates with Ollama using HTTP requests. Ollama receives the prompt, processes the request, and sends back the generated JSON response. Since everything runs locally, response times are short and the system does not depend on an external internet connection after setup.

Installation and Startup of Ollama and n8n

Both Ollama and n8n are deployed using Docker Desktop, which makes the setup easier and more consistent. For the system to work, both containers must be running at the same time. First, Docker Desktop must be installed and running. After that, the existing containers can be checked using Docker Desktop or the command line.

To start Ollama, the container can be launched from Docker Desktop or by running “`docker start ollama`”. When Ollama is running correctly, its API is available on port 11434. This can be verified by opening the `/api/tags` endpoint, which returns a JSON response showing that the service is active.

Next, the n8n container is started in the same way, using Docker Desktop or “`docker start n8n`”. Once running, the n8n editor can be accessed in a browser at port 5678. n8n connects to Ollama through the internal Docker network, not through localhost.

To avoid restarting everything manually, both containers can be configured to start automatically when Docker Desktop launches.

Summary

In this setup, n8n works as the central automation platform, while the AI Agent handles the interpretation of natural language commands. Ollama provides a local and controllable AI backend that generates structured JSON outputs. Running everything in Docker ensures that the system is easy to reproduce, stable, and independent of external AI services. This makes the solution practical and easy to extend for future automation use cases.

Python Automation Engine ('Spracherkennung.py')

This chapter explains the internal mechanisms of the Python Automation Engine. It details how the system handles data ingestion, task routing, browser interaction, and localization. We use the Login Process as a primary example to demonstrate how a command flows through the system.

Prerequisites & Setup

1.1 Before running any code, ensure the following software is installed on your computer:

- Google Chrome Browser: Requirement: Latest stable version.
- Python: Requirement: version 3.11+

1.2 Python Environment (The Automation Core)

The Python script ('0901-nextorder-EngDE.py') requires specific libraries to control the browser and run the web server. Open your terminal in the project folder and run:

```
```bash
pip install flask selenium webdriver-manager
· flask: Runs the local HTTP server to receive commands from n8n.
· selenium: The core library for browser automation.
· webdriver_manager: Automatically downloads and manages the correct ChromeDriver for
your installed Chrome version (no manual path configuration needed).
```

### Layer 1: The Dispatcher (Routing Logic)

1.1 Function: `dispatch\_action(data\_wrapper)`

This layer acts as the “Controller”. It receives the raw JSON command from the AI Agent (via n8n), validates the action type, and routes it to the correct function in the Driver layer.

1.2 Code Logic: The dispatcher checks the `action` key in the JSON payload. If the action is `"login"`, it extracts the `username` and `password` parameters and calls the robot's login method.

1.3 Python code:

```
def dispatch_action(command_json):
 action = command_json.get("action")
 params = command_json.get("params", {})
 # Routing the command
 if action == "login":
 # Pass parameters to the Driver Layer
 robot.login(
 username=params.get("username"),
 password=params.get("password")
)
```

## Layer 2: The Driver (Execution Logic)

### 1.1 Class: SapRobot Method: login(username, password)

This layer acts as the Executor. It contains the low-level Selenium code that interacts directly with the browser's DOM (Document Object Model). It handles the specific steps required to perform the action on the SAP website.

### 1.2 Code Logic: The login function handles the entire browser interaction sequence:

Initialize: Starts the Chrome driver if it's not running.

Navigate: Opens the SAP Fiori URL.

Interact: Waits for the username/password fields to load, types the credentials, and clicks the "Log On" button.

Verify: Waits for the SAP Header Logo to ensure the login was successful.

### 1.3 Python code:

```
def login(self, username, password):
 # 1. Open the SAP Fiori URL
 self.driver.get('https://aimprd.advapp.de/sap/bc/ui2/flp')
 # 2. Input Credentials (using Selenium Explicit Waits)
 user_field = self.wait.until(EC.presence_of_element_located((By.ID,
 'USERNAME_FIELD')))
 user_field.send_keys(username)
 pass_field = self.driver.find_element(By.ID, 'PASSWORD_FIELD')
 pass_field.send_keys(password)
 # 3. Trigger Login
 self.driver.find_element(By.ID, 'LOGIN_LINK').click()
 # 4. Wait for Success Indicator
 self.long_wait.until(EC.presence_of_element_located((By.ID, 'shell-header-logo')))
```

## Localization: Bilingual Interface Support

### Location: SapRobot Class (action\_map, phase\_map, app\_translations)

The system is designed to run on both English and German SAP instances without code modification.

The Logic: Instead of hardcoding XPath text (e.g., text()='Start'), the system generates XPaths dynamically using mapping dictionaries.

### 4.1 Operation Mapping: Maps business intent to UI labels.

Python code:

```
action_map = {
 "start": ["Start", "starten", "Starten"],
 "end": ["Finish", "End", "beenden", "abschließen"]
}
phase_map = {
 "setup": ["Setup", "Rüsten"],
 "processing": ["Processing", "Bearbeitung"]
}
```

Result: When the AI sends action: "start", the bot searches for buttons labeled "Start Setup", "Rüsten starten", "Start Processing", etc.

4.2 App Name Mapping: In `open_app`, if the bot cannot find a tile named "Capture of production data" (EN), it automatically searches for "Erfassung von Produktionsdaten" (DE).

## How to Run

Start the Server: Run the script 'Spracherkennung.py' in a terminal:

Wait for the message:  SapRobot Ready. Flask Server running...

Connect n8n: Ensure your n8n HTTP Request node points to:

URL: <http://localhost:5000/execute>

Method: POST

Body: Raw JSON (from Ollama)

Verify Connection: When n8n sends a request, the Python terminal will log:

```
Received data wrapper: {...} ▶ Dispatching: start_order...
```

## Summary of Architecture

This two-layer pattern (Dispatch -> Robot Method) is applied to all system actions:

Login: if `action == "login"` -> `robot.login()`

Open App: if `action == "open_app"` -> `robot.open_app()`

Example use cases:

JSON Action	Python Method	Description
<code>open_app</code>	<code>robot.open_app(...)</code>	Navigates to a Fiori tile. Handles app name translation.
<code>start_order</code>	<code>robot.process_order_step(..., action='start')</code>	Starts a production phase (Setup/Processing).
<code>end_order</code>	<code>robot.process_order_step(..., action='end')</code>	Finishes a production phase.

This structure ensures that the Business Logic (what to do) is decoupled from the Technical Implementation (how to click), making the code modular and easy to maintain

## Installation auf Linux-Rechner

Auf dem Linux-Rechner hat die Lösung nicht zuverlässig funktioniert, da mehrere Komponenten stark vom lokalen System und der Netzwerkumgebung abhängig sind. Besonders kritisch war Selenium, da der Browser und der passende WebDriver korrekt zueinander passen und sauber starten müssen. Unter Linux kam es dabei häufiger zu Problemen durch fehlende Abhängigkeiten, Rechte-Themen beim Ausführen und Unterschiede im Setup von Chrome bzw. Chromium, wodurch der automatisierte Browserstart instabil wurde oder gar nicht erst funktionierte.

Zusätzlich hat die Kommunikation zwischen n8n, Whisper und dem Python-Server unter Linux nicht immer wie erwartet funktioniert, da Docker-Netzwerke, Port-Freigaben und Hostnamen je nach System unterschiedlich greifen. Dadurch kamen Webhook-Requests teilweise nicht an oder einzelne Services waren aus n8n heraus zeitweise nicht erreichbar. Insgesamt war unter Linux mehr Systemkonfiguration und Debugging nötig, um Browserautomation und Container-Kommunikation stabil zum Laufen zu bringen.

Außerdem blieb die Installation bei uns nicht dauerhaft stabil. Obwohl zunächst alles eingerichtet war, mussten wir nach dem Schließen aller Programme, einem Herunterfahren und dem nächsten Start vieles erneut einrichten oder neu installieren, zum Beispiel den WebDriver für Selenium. Auch ngrok war zwar installiert, war beim nächsten Start aber teilweise nicht mehr verfügbar oder nicht mehr korrekt eingebunden, sodass eine erneute Installation bzw. Konfiguration nötig war. Dadurch war der Workflow unter Linux für uns nicht zuverlässig reproduzierbar und kostete deutlich mehr Zeit als unter Windows.

## Fazit

Abschließend lässt sich festhalten, dass der gewählte Aufbau eine praxistaugliche Grundlage für sprachgesteuerte Prozessschritte in n8n bietet. Durch die Kombination aus stabiler Container-Umgebung, klarer Schnittstelle zum Python-Robot-Server und robuster Sprachvorverarbeitung konnten typische Fehlerquellen wie Fehlhörer, unterschiedliche Schreibweisen oder unvollständige Eingaben deutlich reduziert werden. Insgesamt zeigt das System, dass sich wiederkehrende Bedienhandlungen in SAP zuverlässig automatisieren lassen, sofern die Schnittstellen sauber definiert sind und die Sprachbefehle konsequent in ein einheitliches Kommandoformat überführt werden.

Zusätzlich ist die Lösung gut erweiterbar. Weitere Anwendungsfälle können ohne großen Aufwand ergänzt werden, da die Sprachlogik bereits robust aufgebaut ist und neue Befehle in der Regel nur als zusätzliche Regeln oder Muster aufgenommen werden müssen. Der Prompt im AI-Agenten kann leicht angepasst oder erweitert werden, sodass der Agent mehrere Use Cases zuverlässig erkennt und in das passende JSON-Format übersetzt. Parallel dazu lässt sich auch der Python-Code modular erweitern, indem neue Aktionen oder Prozessschritte ergänzt werden. Dadurch bleibt das System flexibel und kann schrittweise an weitere Anforderungen aus der Produktion angepasst werden.