Hochschule Bielefeld
University of Applied Sciences and Arts

# Entwicklung eines Tools zum automatisierten Fine-Tuning lokaler Large Language Models für kundenspezifische TwinCAT-Structured-Text-Anwendungen

**Bachelorarbeit**

**Nicht freigegeben**


| | |
|---|---|
| Verfasser: | **Enzo Zacharias**<br>Pfirsichweg 6A<br>33334 Gütersloh |
| Matrikelnummer: | 1321700 |
| Studiengang: | Digitale Technologien |
| Fachbereich: | Ingenieurwissenschaften und Mathematik |
| Semester: | 7 |
| Erstprüfer: | Prof. Dr. rer. nat. Stefan Berlik |
| Zweitprüfer: | Jannis Doppmeier |
| Eingereicht am: | 16.12.2025 |

In Kooperation mit der Beckhoff Automation GmbH & Co. KG
Hülshorstweg 20, 33415 Verl

## Sperrvermerk

Diese wissenschaftliche Arbeit enthält vertrauliche Daten der Beckhoff Automation GmbH & Co. KG. Die Weitergabe, Veröffentlichung, Vervielfältigung der wissenschaftlichen Arbeit oder die Nutzung der Inhalte der wissenschaftlichen Arbeit zu Forschungszwecken – auch nur auszugsweise – ist ohne ausdrückliche Genehmigung der Beckhoff Automation GmbH & Co. KG nicht gestattet. Dies betrifft insbesondere, aber nicht ausschließlich auch die Aufnahme der wissenschaftlichen Arbeit in eine Bibliothek der Hochschule, die kommerzielle Nutzung, das Laden der wissenschaftlichen Arbeit auf Servern von Dritten oder die Überprüfung mit Hilfe von auf externen Servern laufender Plagiatssoftware. Sofern ein Programmcode (im Folgenden: Code) im Rahmen der wissenschaftlichen Arbeit erstellt wird, umfassen die vertraulichen Daten der Beckhoff Automation GmbH & Co. KG auch diesen Code. Insofern beinhaltet die wissenschaftliche Arbeit lediglich Auszüge des Codes. Sofern erforderlich, wird den Prüfenden bzw. Korrigierenden auf Wunsch eine Einsichtnahme in den vollständigen Code gewährt. Eine Herausgabe des Codes ist nicht gestattet. Die wissenschaftliche Arbeit darf ausschließlich den Prüfenden bzw. den Korrigierenden sowie den Mitgliedern des Prüfungsausschusses bzw. den Prüfungsbeauftragten zugänglich gemacht werden.

## Gleichstellungsvermerk

Die Inhalte der vorliegenden Arbeit beziehen sich in gleichem Maße auf alle Geschlechter (m/w/d). Aus Gründen der besseren Lesbarkeit wird jedoch die männliche Form für alle Personenbezeichnungen gewählt. Es wird an dieser Stelle darauf hingewiesen, dass die ausschließliche Verwendung der männlichen Form geschlechtsunabhängig verstanden werden soll.

## Markenvermerk

Beckhoff® und TwinCAT® sind eingetragene und lizenzierte Marken der Beckhoff Automation GmbH.

NVIDIA© ist eine eingetragene Marke der NVIDIA Corporation.

## Danksagung

Diese Bachelorarbeit ist im Rahmen des praxisintegrierten Studiengangs Digitale Technologien am Campus Gütersloh der Hochschule Bielefeld entstanden. Mein besonderer Dank gilt allen Personen, die mich während der gesamten Studienzeit und insbesondere im Zuge dieser Arbeit begleitet und unterstützt haben.

Mein Dank gilt der Beckhoff Automation GmbH & Co. KG sowie Hans Beckhoff für die Möglichkeit, diese Arbeit im Unternehmen umzusetzen. Ebenso danke ich dem Talent Management für die kontinuierliche Unterstützung und den offenen, unkomplizierten Austausch.

Herrn Professor Berlik danke ich für die Betreuung als Erstprüfer, die fachlichen Impulse und das hilfreiche Feedback, sowohl in dieser Arbeit als auch in vorangegangenen Projektarbeiten. Die Betreuung war jederzeit konstruktiv und angenehm.

Mein besonderer Dank gilt zudem Jannis Doppmeier als Zweitprüfer und fachlichem Betreuer im Unternehmen. Seine konstruktiven Hinweise und das Korrekturlesen der Arbeit waren für den Fortschritt des Projekts sehr wertvoll. Besonders geschätzt habe ich den Freiraum, den er mir bei der Umsetzung gelassen hat, wodurch ich viele eigene Ideen einbringen konnte. Vielen Dank für die persönliche Begleitung über mehr als zwei Jahre.

Ebenso danke ich Benjamin Jurke für seine Unterstützung bei technischen Fragen sowie für seine maßgebliche Hilfe bei der Bereitstellung der erforderlichen Hardware, die für die Umsetzung des Projekts von großer Bedeutung war.

Für die vielen alltäglichen Anregungen und die Unterstützung bei fachlichen Fragen danke ich meinen Kolleginnen und Kollegen Michael Zeunert, Robin Klett und Antoine Angert. Darüber hinaus bedanke ich mich bei Timo Habighorst für seine wertvolle Hilfe bei der Integration interner Tools.

Mein Dank richtet sich außerdem an meine Freunde und meine Familie, die mir in dieser intensiven Phase stets Rückhalt gegeben haben. Ihre Unterstützung, ihre Ermutigung und ihr Zuspruch haben maßgeblich dazu beigetragen, dass ich diese Zeit erfolgreich bewältigen konnte.

Der Zuspruch, die Unterstützung und das entgegengebrachte Vertrauen haben maßgeblich dazu beigetragen, diese Herausforderung erfolgreich zu meistern. Mein Dank gilt allen, die mich auf diesem Weg begleitet haben.

## Kurzfassung

Die Bachelorarbeit beschreibt die Entwicklung eines Tools zum automatisierten Fine-Tuning lokaler Large Language Models für TwinCAT-Structured-Text-Anwendungen in Kooperation mit Beckhoff Automation GmbH & Co. KG. Ausgangspunkt ist der Bedarf an datenschutzkonformen, lokal betreibbaren LLMs bei gleichzeitig fehlenden öffentlich verfügbaren ST-Trainingsdaten. Das Tool gliedert sich in vier gekapselte Teilbereiche. Dazu zählen die semantische Analyse realer TwinCAT-Projekte, die darauf basierende Generierung synthetischer ST-Trainingsdaten, das ressourceneffiziente Fine-Tuning ausgewählter LLMs mittels LoRA sowie ein TwinCAT-spezifisches Benchmarking mit Compilerprüfungen und Unit-Tests. Auf Basis eines Unternehmensprojekts wird ein synthetischer Datensatz mit rund 5.000 Beispielen aufgebaut und zur Anpassung mehrerer Qwen-Modelle eingesetzt. Die Evaluation zeigt im Vergleich zu den jeweiligen Ausgangsmodellen auf dem TwinCAT-spezifischen Benchmark eine deutliche Steigerung der syntaktischen und semantischen Qualität der Codegenerierung um durchschnittlich 18,0 % bzw. 15,9 %. Ein Vergleich mit Cloud-Modellen verdeutlicht, dass lokales Fine-Tuning die Leistungsunterschiede spürbar reduziert, ohne Vertraulichkeitsanforderungen zu verletzen. Die Arbeit stellt damit eine durchgängige Lösung bereit, die LLMs an die domänenspezifischen Anforderungen der TwinCAT-Programmierung anpasst.

## Abstract

This thesis presents a tool for automated fine-tuning of local large language models for TwinCAT Structured Text applications in cooperation with Beckhoff Automation GmbH & Co. KG. It addresses the need for privacy-preserving, locally deployable models while public ST training data are largely unavailable. The tool is divided into four encapsulated components. These comprise the semantic analysis of real TwinCAT projects, the generation of synthetic ST training data based on this analysis, the resource-efficient fine-tuning of selected LLMs using LoRA, and a TwinCAT-specific benchmarking pipeline with compiler checks and unit tests. A company project is used to construct a synthetic dataset with around 5,000 examples that serves as the basis for adapting several Qwen models. The evaluation shows, compared to the respective base models on the TwinCAT-specific benchmark, an average increase in syntactic and semantic code quality of 18.0 % and 15.9 %. A comparison with cloud models demonstrates that targeted local fine-tuning can substantially reduce the performance gaps without violating confidentiality requirements and thus aligns LLMs with the domain-specific needs of TwinCAT programming.

## Inhaltsverzeichnis

## Abbildungsverzeichnis



## Abbildungsverzeichnis Anhang

## Tabellenverzeichnis



## Tabellenverzeichnis Anhang

## Listingverzeichnis Anhang

## Abkürzungs- und Symbolverzeichnis

| Abkürzung | Bedeutung |
|---|---|
| API | Application Programming Interface |
| BPE | Byte Pair Encoding |
| DLL | Dynamic-Link Library |
| EVA | Eingabe-Verarbeitung-Ausgabe |
| FB | Funktionsblock |
| FFN | Feed-Forward Network |
| GGUF | Kompaktes, quantisiertes Modellformat |
| IEC 61131-3 | Norm für SPS-Programmiersprachen der internationalen elektrotechnischen Kommission |
| JSON | JavaScript Object Notation |
| LLM | Large Language Model |
| LoRA | Low-Rank Adaption (PEFT-Methode) |
| PEFT | Parameter-Efficient Fine-Tuning |
| RAG | Retrieval-Augmented Generation |
| ST | Structured Text |
| TcDUT | TwinCAT-Data Unit Type |
| TcGVL | TwinCAT-Global Variable List |
| TcPOU | TwinCAT-Program Organization Unit |
| TwinCAT | The Windows Control and Automation Technology |
| UTF-8 | 8-Bit Unicode Transformation Format |
| VRAM | Video Random Access Memory |
| XML | Extensible Markup Language |

| Symbol | Bedeutung |
|---|---|
| $y$ | Ausgabe des Neurons |
| $f(\cdot)$ | Aktivierungsfunktion |
| $b$ | Bias-Term eines Neurons |
| $x$ | Eingabevektor eines Neurons |
| $x_i$ | $i$-te Komponente des Eingabevektors |
| $w_i$ | Gewicht zu Eingabe $x_i$ |
| $n$ | Anzahl der Eingabekomponenten / Dimension eines Neurons |
| $d$ | Dimension eines Embedding-Vektors |
| $W$ | Ursprüngliche Gewichtsmatrix |
| $W'$ | Angepasste Gewichtsmatrix |
| $\Delta W$ | Low-Rank-Update |
| $A, B$ | LoRA-Matrizen |
| $r$ | Rang der LoRA- Matrizen |
| $\alpha$ | LoRA-Stabilisierungsfaktor |
| $h$ | Ausgabevektor nach LoRA |
| $xAB$ | Trainierter LoRA-Korrekturterm |
| $Q$ | Query-Matrix |
| $K$ | Key-Matrix |
| $V$ | Value-Matrix |
| $W^O$ | Ausgangsprojektion |
| $W_i^Q$ | Query-Projektionsmatrix für Head $i$ |
| $W_i^K$ | Key-Projektionsmatrix für Head $i$ |
| $W_i^V$ | Value-Projektionsmatrix für Head $i$ |
| $head_i$ | Ausgabe des $i$-ten Attention-Heads |
| $h$ | Anzahl der Attention-Heads |

| | |
|---|---|
| $Concat(\cdot)$ | Führt mehrere Heads zu einer Ausgabe zusammen |
| $softmax(\cdot)$ | Normalisierte Werte zu einer Wahrscheinlichkeitsverteilung |
| $Attention(\cdot)$ | Berechnet gewichtete Abhängigkeiten zwischen Tokens |
| $MultiHead(\cdot)$ | Kombiniert mehrere parallele Attention-Heads |
| $FFN(\cdot)$ | Position-weises Feed-Forward-Netzwerk im Transformer |

# 1 Einleitung

Die vorliegende Bachelorarbeit wird am Campus Gütersloh der Hochschule Bielefeld im Studiengang Digitale Technologien verfasst und in Kooperation mit der Beckhoff Automation GmbH & Co. KG durchgeführt. Gegenstand der Arbeit ist die Entwicklung eines Tools, das die automatisierte Analyse von TwinCAT-Projekten ermöglicht und auf dieser Grundlage synthetische Trainingsdaten erzeugt, um lokale Large Language Models (LLMs) gezielt für den Einsatz in der Automatisierungsbranche anzupassen. Abschnitt 1.1 erläutert die Motivation für die Bearbeitung des Themas. In Abschnitt 1.2 werden die Ziele der Arbeit präzisiert, bevor Abschnitt 1.3 die inhaltliche und formale Struktur der Arbeit darstellt.

## 1.1 Einführung und Motivation

Die Beckhoff Automation GmbH & Co. KG, im Folgenden als das betrachtete Unternehmen bezeichnet, integriert generative künstliche Intelligenz in die Entwicklungsumgebung TwinCAT, um SPS-Programmierer bei der Erstellung von Steuerungsprojekten zu entlasten und die Effizienz der Softwareentwicklung zu steigern [BECK25j]. Damit reagiert das betrachtete Unternehmen auf die wachsende Bedeutung von LLMs im industriellen Umfeld, die zunehmend als Werkzeuge zur Unterstützung im Entwicklungsprozess eingesetzt werden. Parallel dazu äußern Kunden den Wunsch nach lokal betriebenen LLMs, die unabhängig von Cloud-Anbietern genutzt werden können und gleichzeitig Datenschutz- und Compliance-Anforderungen erfüllen. Zudem steigt die Nachfrage nach LLMs, die nicht nur generische Programmieraufgaben lösen, sondern speziell auf kundenspezifische Projekte zugeschnitten sind und in der Lage sind, Code zu generieren, der den branchenspezifischen Anforderungen gerecht wird.

In der Praxis stoßen allgemeine Open-Source-LLMs jedoch schnell an Grenzen. Lokale LLMs können zwar ressourcenschonend betrieben werden, besitzen jedoch aufgrund einer geringeren Anzahl an Parametern nur eingeschränkte Wissensbestände. Daher verfügen sie weder über ausreichende Kenntnisse der Programmiersprache Structured Text (ST) noch über ein fundiertes Verständnis für Beckhoff-spezifische Bibliotheken oder etablierte Konstruktionsprinzipien. Ohne gezieltes Fine-Tuning sind lokale LLMs nicht in der Lage, qualitativ hochwertigen ST-Code zu generieren. Das Fine-Tuning lokaler LLMs wird zusätzlich dadurch erschwert, dass für ST kaum öffentliche Datensätze verfügbar sind und reale Kundenprojekte aus Datenschutzgründen nicht als Trainingsgrundlage verwendet werden können.

Vor diesem Hintergrund gewinnt die Generierung synthetischer Trainingsdaten zentrale Bedeutung. Synthetische Daten können gezielt so entworfen werden, dass sie typische Strukturen industrieller Steuerungsprogramme, spezifische Beckhoff-Bibliotheken sowie kundenspezifische Richtlinien berücksichtigen. Da die Qualität der Trainingsdaten maßgeblich über den Erfolg des Fine-Tunings entscheidet, ist die systematische Validierung unerlässlich. Parser- und Compiler-gestützte Prüfmechanismen können sicherstellen, dass ausschließlich syntaktisch korrekte Daten genutzt werden, womit die Notwendigkeit eines Tools mit integrierten Validierungsmechanismen deutlich wird.

Für das betrachtete Unternehmen ergibt sich daraus die Chance, interne wie auch kundenspezifische LLMs bereitzustellen, die optimal an die Anforderungen der Automatisierungsbranche angepasst sind und den Entwicklungsprozess unterstützen. Die Arbeit trägt dazu bei, die Lücke zwischen generischen LLMs und den domänenspezifischen Anforderungen der TwinCAT-Programmierung zu schließen und schafft damit die Grundlage für ein gezieltes Fine-Tuning lokaler LLMs.

## 1.2 Zielsetzung

Im Rahmen dieser Arbeit soll in einem Bearbeitungszeitraum von zwölf Wochen ein Tool entwickelt werden, das die automatisierte Analyse von TwinCAT-Projekten mit der Generierung synthetischer Trainingsdaten verbindet und diese für das Fine-Tuning lokaler LLMs nutzbar macht. Dazu soll das Tool TwinCAT-Projekte systematisch einlesen und auf relevante Strukturmerkmale sowie Designprinzipien untersuchen, sodass die analysierten Informationen die gezielte Steuerung der Datengenerierung ermöglichen. Die erzeugten Trainingsdaten sollen bereits während der Erstellung durch parserbasierte Prüfungen kontrolliert und mithilfe von Feedbackschleifen kontinuierlich verbessert werden. Für das Fine-Tuning soll das Tool ein ressourcenschonendes Verfahren integrieren, damit hochparametrisierte LLMs angepasst werden können. Die Validierung soll sowohl syntaktisch durch Parser als auch semantisch mithilfe von Unit-Tests erfolgen, um die Qualität der angepassten LLMs zu bewerten und den Erfolg des Fine-Tunings messbar zu machen. Abschließend sollen die trainierten LLMs durch das Tool für Testinferenz bereitgestellt und zentral verwaltet werden, wodurch eine unmittelbar nutzbare Lösung entsteht.

## 1.3 Aufbau der Arbeit

In Kapitel 2 werden die wissenschaftlichen Grundlagen vorgestellt, die für das Verständnis der Arbeit erforderlich sind. Dazu zählen neben der Programmiersprache Structured Text auch die Funktionsweise von LLMs, Techniken des Fine-Tunings, Compilertechnologien sowie Verfahren zur Generierung synthetischer Trainingsdaten. Kapitel 3 widmet sich der Analyse der bestehenden Herausforderungen und leitet daraus konkrete Anforderungen an die Lösung ab. In Kapitel 4 wird der aktuelle Stand der Technik diskutiert, bevor Kapitel 5 das Lösungskonzept entwickelt und die konzeptionelle Gestaltung des Tools beschreibt. Kapitel 6 stellt die praktische Umsetzung dar und erläutert die einzelnen Schritte zur Realisierung des Tools, wobei zugleich die Evaluation der Ergebnisse erfolgt. Kapitel 7 fasst abschließend die Ergebnisse zusammen und gibt einen Ausblick auf mögliche Weiterentwicklungen.

# 2 Wissenschaftliche Grundlagen

Dieses Kapitel erläutert die theoretischen und technischen Grundlagen, die für das Verständnis dieser Arbeit erforderlich sind. Zunächst wird in Kapitel 2.1 die Programmiersprache Structured Text im Kontext der SPS-Architektur beschrieben, anschließend in Kapitel 2.2 der Aufbau und die Funktionsweise von LLMs auf Basis der Transformer-Architektur. In Kapitel 2.3 folgen die wissenschaftlichen Grundlagen zu Methoden des Fine-Tunings von LLMs, während Kapitel 2.4 die Compiler-Technologien zur syntaktischen Analyse von ST darstellt. Abschließend wird in Kapitel 2.5 die Bedeutung synthetischer Trainingsdaten für domänenspezifisches Fine-Tuning erläutert.

## 2.1 Structured Text im Kontext der SPS-Architektur

Speicherprogrammierbare Steuerungen (SPS) arbeiten zyklusorientiert nach dem Eingabe-Verarbeitung-Ausgabe-Prinzip (EVA). Dabei werden zunächst alle Eingänge eingelesen und durch das Programm verarbeitet. Das Programm analysiert die Eingänge, trifft logische Entscheidungen und aktualisiert auf dieser Basis die Ausgänge. Anschließend beginnt der Zyklus von vorn, sodass die SPS mehrmals pro Sekunde neue Eingangssignale einlesen und Ausgangssignale setzen kann. [WEIN25]

The Windows Control and Automation Technology (TwinCAT) als PC-basierte SPS-Laufzeitumgebung orientiert sich an diesem Grundprinzip, bietet jedoch durch ein taskbasiertes Laufzeitsystem erweiterte Möglichkeiten. Zykluszeiten können festgelegt werden, in denen Programme und Bausteile deterministisch ausgeführt werden. TwinCAT führt in jedem Zyklus das EVA-Prinzip aus. Damit wird ein deterministisches Zeitverhalten sichergestellt, sodass die Steuerung im Sinne der DIN 44300 echtzeitfähig arbeitet. [BECK25a] [BECK25b]

Die Norm IEC 61131-3 definiert fünf standardisierte Programmiersprachen zur Programmierung von SPS-Systemen. TwinCAT unterstützt alle in der Norm definierten Sprachen, nämlich Kontaktplan, Funktionsplan, Ablaufsprache, Anweisungsliste sowie Structured Text (ST). In TwinCAT werden alle Sprachen durch einen Compiler in einen einheitlichen, maschinenlesbaren Code übersetzt. [IEC25, S. 9] [BECK25d]

ST ist eine hochsprachenähnliche, textuelle Sprache, die prozedurale und objektorientierte Programmierung ermöglicht. Der Code besteht aus Anweisungen, die in Kontrollstrukturen wie Bedingungen und Schleifen organisiert werden können. ST stellt sprachliche Mittel (z. B. Schleifen, Verzweigungen) bereit, um komplexe Algorithmen und Berechnungen abzubilden.

Dadurch wird die Übersichtlichkeit erhöht und Fehlerquellen reduziert. In TwinCAT-Projekten wird ST häufig bevorzugt, um komplexe Logik abzubilden. [BECK25c]

Die IEC 61131-3 definiert Software-Bausteine als Program Organization Units (POUs), die in TwinCAT für alle Sprachen implementierbar sind. Die drei POU-Typen sind Programme, Funktionsbausteine und Funktionen. Ein Programm beschreibt die oberste Ablauf-Einheit einer SPS-Steuerung und koordiniert den Hauptsteuerungsablauf. Programme und die dazugehörigen Variablen können zyklusübergreifend Zustände behalten. Ein Funktionsbaustein (FB) ist eine wiederverwendbare Bausteineinheit mit Ein- und Ausgangsvariablen sowie internem Zustand. Ein FB kann in einem Programm beliebig oft instanziiert und mit unterschiedlichen Variablen verknüpft werden. Die Besonderheit besteht im internen Speicher. Aufeinanderfolgende Aufrufe mit identischen Eingabewerten liefern nicht zwingend identische Ergebnisse, da diese vom gespeicherten Zustand abhängen. Bei jedem Aufruf wird die interne Logik des Funktionsbausteins ausgeführt und die Ausgänge aktualisiert. Eine Funktion ist demgegenüber im Wesentlichen ein Unterprogramm mit Rückgabewert. Diese besitzt keinen internen Speicher und ist daher zustandslos, das heißt, gleiche Eingaben führen typischerweise zu gleichen Ausgaben. [IEC25, S.77-134] [CONT25] [BECK25e] [BECK25f] [BECK25g]

Für die Entwicklung effizienten Programmcodes und die Wiederverwendung erprobter Logik wird in TwinCAT auf Bibliotheken zurückgegriffen. TwinCAT enthält eine umfangreiche Standardbibliothek, die alle in IEC 61131-3 definierten POUs bereitstellt (z. B. Timer, Zähler, mathematische Funktionen). Weitere spezialisierte Bibliotheken decken verschiedene Anwendungsgebiete ab. Der Aufruf von Bibliotheksfunktionen und -bausteinen erfolgt äquivalent zu normalen Funktionen und Funktionsbausteinen, sobald die entsprechende Bibliothek in das Projekt eingebunden ist. Der Einsatz von Bibliotheken fördert effizienten, wartbaren Code, da bewährte Bausteine direkt genutzt werden können. [IEC25, S.77-134] [FECH25] [BECK25h]

Die Qualität von ST kann durch syntaktische und semantische Prüfungen bewertet werden. Parser-Tools, die IEC-61131-3-kompatibel sind, werden für die syntaktische Analyse von ST verwendet. Diese prüfen die Einhaltung der Grammatikregeln und melden Syntaxfehler bei Regelverstößen. Die Funktionsweise von Parsern wird in Kapitel 2.4 detailliert beschrieben. Zur Beurteilung der semantischen Qualität können Unit-Tests implementiert werden. Ein Unit-Test prüft einzelne funktionale Einheiten (z. B. einen Funktionsbaustein oder eine Funktion), indem vordefinierte Eingaben angelegt und die Ausgaben mit erwarteten Ergebnissen abgeglichen werden. Unterschiedliche Testszenarien, einschließlich praxisnaher Eingabewerte, Grenzwerte und Sonderfälle, ermöglichen eine umfassende Bewertung der semantischen Qualität. [HENNE25]

## 2.2 Large Language Models

LLMs sind neuronale Sprachmodelle, die in natürlicher Sprache mit Anwendern interagieren. Ein Sprachmodell ordnet einer Wortsequenz eine Wahrscheinlichkeit zu und kann durch Generieren neuer Wörter aus dieser Verteilung Texte erzeugen. LLMs unterscheiden sich von früheren Ansätzen wie n-Gramm-Modellen durch eine erheblich größere Parameterzahl und den Einsatz neuronaler Netze. Trainiert werden tiefe neuronale Netzwerke darauf, das nächste Wort aus einem gegebenen Kontext vorherzusagen, wobei statistische Zusammenhänge aus großen Textmengen implizit erlernt werden. Das Training auf umfangreichen Textkorpora führt zu Wahrscheinlichkeitsverteilungen der Sprache und ermöglicht die Generierung zusammenhängender Texte. [JURA25a]

Der erste Verarbeitungsschritt vom Text zur Modelleingabe ist die Tokenisierung. Dabei wird ein Eingabetext in eine Folge von Tokens zerlegt. Tokens können aus Wörtern, Wortbestandteilen oder einzelnen Zeichen bestehen. Eine Herausforderung entsteht bei der Festlegung der Token-Einheiten, da Sonderfälle wie Kontraktionen wie *I'm* oder Wortabgrenzungen durch Leerzeichen auftreten. Für robuste Verfahren werden sublexikalische Tokens anstelle vollständiger Wörter verwendet. Sublexikalische Tokens bilden die Bausteine eines Wortes ab, etwa Buchstaben, Silben oder Morpheme [HASE25]. Die Tokenisierung erfolgt mit Verfahren wie Byte-Pair Encoding (BPE) oder dem Unigram-Modell. Beide Algorithmen lernen aus großen Textkorpora ein Vokabular häufiger Teilsequenzen, sodass seltene oder unbekannte Wörter in sinnvolle Teilwörter zerlegt werden. BPE startet mit einzelnen Zeichen und führt iterativ das häufigste benachbarte Tokenpaar zu einem neuen Token zusammen. Nach genügend vielen Zusammenführungen entsteht ein Vokabular, in dem häufige Wörter als einzelnes Token repräsentiert sind, während seltene oder neue Wörter in mehrere Teilwort-Tokens aufgespalten werden. Auch Programmcode lässt sich auf diese Weise tokenisieren, allerdings mit zusätzlichen Anforderungen, da eigene Token-Arten wie Schlüsselwörter, Bezeichner oder Einrückungen vorkommen. Moderne Tokenizer berücksichtigen diese Besonderheiten und optimieren den Prozess entsprechend. Die Tokenisierung stellt eine konsistente Folge diskreter Einheiten für die Modelleingabe bereit. Jedes Token wird im Vokabular durch eine eindeutige ID repräsentiert. [JURA25b]

Da neuronale Netze mit numerischen Vektoren arbeiten, müssen Tokens als Vektoren dargestellt werden. Die einfachste numerische Darstellung ist ein One-Hot-Vektor mit der Dimension der Vokabulargröße und einem Eintrag 1 an der Position der Token-ID. Diese Darstellung ist bei großen Vokabularen ineffizient, weil die Vektoren stark spärlich besetzt sind. Üblicher sind Embeddings, also gelernte d-dimensionale dichte Vektorrepräsentationen. Jeder Token-ID

wird ein Embedding-Vektor zugeordnet, zum Beispiel mit 256 oder 768 Dimensionen. Solche Embeddings kodieren semantische Ähnlichkeiten, sodass ähnliche Wörter im Vektorraum nahe beieinander liegen. Die Embeddings werden während des Trainings mit optimiert. Die endgültige Vektorrepräsentation entsteht kontextabhängig und ergibt sich in tieferen Modellschichten dynamisch.[JURA25c]

Die eingebetteten Tokensequenzen werden anschließend in einem tiefen neuronalen Netzwerk verarbeitet. Ein künstliches Neuron berechnet eine gewichtete Summe der Eingaben, addiert einen Bias-Term und wendet eine nichtlineare Aktivierungsfunktion an. Formal ergibt sich die Ausgabe eines Neurons $y$ mit Eingabevektor $x$ ausdrücken als:

$$y = f(b + \sum_{i=1}^{n} w_i x_i) \tag{2.1}$$

Dabei bezeichnen $w_i$ die gelernten Gewichte der Eingabe $x_i$, $b$ den Bias und $f$ eine Aktivierungsfunktion wie Sigmoid-, Tanh, oder ReLU-Funktion. Durch das Stapeln vieler solcher Neuronen in mehreren Schichten entstehen tiefe Netze, die komplexe Funktionen approximieren. Dadurch lassen sich längere Kontexthistorien berücksichtigen und eine bessere Generalisierung auf neue Sequenzen erreichen. Am Ausgang entsteht eine Wahrscheinlichkeitsverteilung über das nächste Token im Vokabular. Wiederholtes autoregressives Auswählen aus diesen Verteilungen ermöglicht die Textgenerierung. Moderne LLMs umfassen Milliarden lernbarer Parameter, beispielsweise GPT-3 mit 175 Milliarden Parametern [ALAC20]. Solche Modelle werden im selbstüberwachten Verfahren auf große Textkorpora trainiert. Ziel ist die Vorhersage der Wahrscheinlichkeit des nächsten Tokens in einer gegebenen Sequenz, sodass statistische Strukturen und semantische Zusammenhänge der Sprache erlernt werden und Textgenerierung möglich wird. [JURA25d]

### 2.2.1 Transformer-Architektur

Moderne LLMs basieren auf der Transformer-Architektur, die seit 2017 von Vaswani et al. definiert ist [VASW17]. Die Abbildung 2.1 veranschaulicht den grundlegenden Aufbau eines Transformer-Modells, das aus einem Encoder- und einem Decoder-Stack besteht.

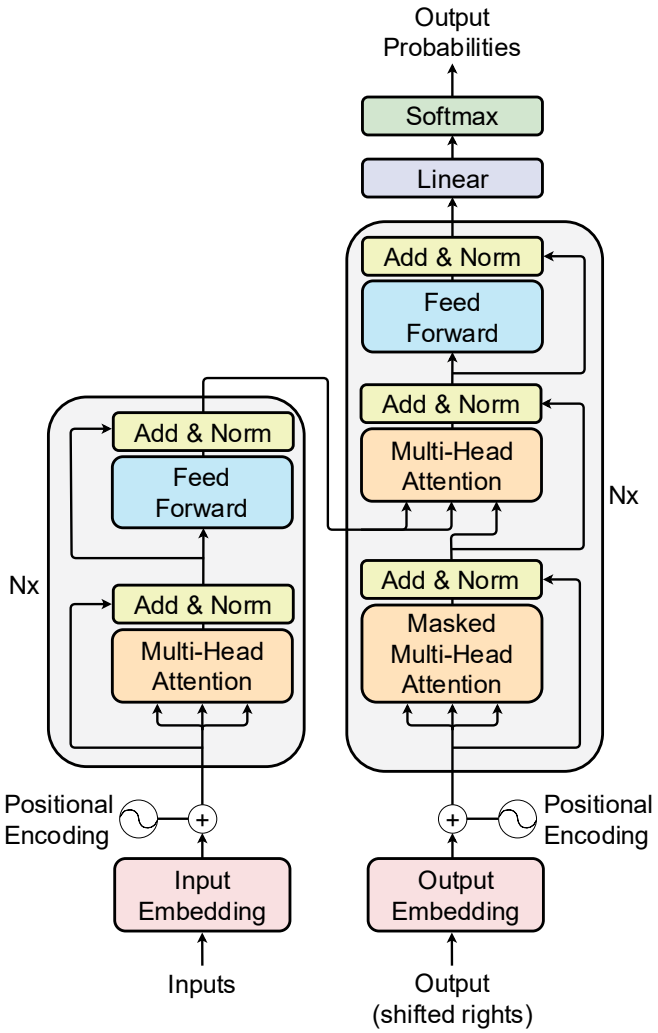


Abbildung 2.1: Transformer-Architektur [VASW17]

Die Transformer-Architektur ist ein neuronales Sequenzmodell, das vollständig auf Attention (Aufmerksamkeitsmechanismen) basiert und auf rekurrente oder konvolutionale Strukturen verzichtet. Kernidee der Architektur ist die Self-Attention (Selbstaufmerksamkeit). Jedes Ausgabe-Token berechnet eine Repräsentation, indem alle Eingabetokens betrachtet und relevante Anteile gewichtet übernommen werden. Dadurch können kontextuelle Beziehungen zwischen weit entfernten Tokens erfasst werden. Das ist für allgemeine Sprache wichtig, um beispielsweise Zuordnungen von Pronomen zu Bezugswörtern über mehrere Sätze hinweg zu ermöglichen. Die Architektur besteht aus einer Eingabe-Einbettung, einer Serie von Transformer-Blöcken und einem Ausgabe-Kopf, der die letzten versteckten Zustände in Wahrscheinlichkeiten über das Vokabular umwandelt. Neben dieser ursprünglichen Form existieren optimierte Varianten der Transformer-Architektur, die in modernen LLMs verstärkt zum Einsatz kommen. Diese Weiterentwicklungen behalten die Kernprinzipien bei und verbessern Aspekte wie Effizienz und Stabilität (vgl. Abbildung A 1). [VASW17] [JURA25e]

Jeder Transformer-Block im Encoder-Stack (linke Seite) enthält zwei Kernsubschichten. Im ersten Schritt eine Multi-Head Self-Attention und danach ein Feed-Forward-Netzwerk (FFN). Im Decoder-Stack hat jeder Block drei Subschichten. Zuerst folgt eine Self-Attention mit Maskierung, danach eine Encoder-Decoder-Attention und abschließend ein FFN. Der Encoder-Stack sowie der Decoder-Stack arbeiten schichtweise. Die Ausgabe der jeweiligen vorherigen Schicht dient als Eingabe der nächsten Schicht. Zusätzliche Residualverbindungen erhalten den Informationsfluss, indem die Eingabe zur Ausgabe addiert wird. Zudem ist eine Layer-Normalisierung zur Stabilisierung des Lernprozesses vorhanden. Residualverbindungen und Layer-Normalisierung werden durch die gekennzeichneten Add-&-Norm-Blöcke angedeutet. [VASW17] [JURA25e]

In der Self-Attention wird jedes Token in Beziehung zu allen anderen Tokens derselben Sequenz gesetzt. Ausgangspunkt dafür ist die Eingabesequenzmatrix

$$X \in \mathbb{R}^{n \times d}, \tag{2.2}$$

wobei n die Anzahl der Tokens und d die Modelldimension bezeichnet. Aus dieser Matrix werden drei unterschiedlichen Projektionen gebildet:

$$Q = XW^Q, \qquad K = XW^K, \qquad V = XW^V \tag{2.3}$$

Die Queries $Q$ geben an, wonach ein Token sucht, die Keys $K$ beschreiben bereitgestellte Eigenschaften, und Values $V$ enthalten Informationen, die in die Ausgabe einfließen. Zur Bestimmung der Stärke der Attention zwischen zwei Tokens wird das Skalarprodukt von $Q_i$ und $K_j$ berechnet. Die Berechnung wird als Scaled Dot-Product Attention bezeichnet und lautet:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.4}$$

Dabei dient $\sqrt{d_k}$ als Skalierungsfaktor, um zu große Werte im Skalarprodukt zu vermeiden. Die Softmax-Normalisierung sorgt dafür, dass die berechneten Attention-Werte als Wahrscheinlichkeiten interpretiert werden können, die sich auf alle Tokens verteilen. Jeder Ausgabevektor stellt eine gewichtete Summe der Value-Vektoren aller Eingabetokens dar, wobei das Gewicht angibt, wie viel Attention Token $i$ dem jeweiligen Token $j$ schenkt. Dieser Mechanismus ermöglicht es dem Modell, bedeutsame Kontextstellen zu fokussieren und weniger relevante Wörter abzuschwächen. [VASW17]

In der Praxis wird diese Berechnung mit mehreren parallelen Attention-Heads durchgeführt, um unterschiedliche Aspekte der Relationen einzufangen. Bei der Multi-Head-Attention werden h unabhängige Attention-Ausgaben berechnet und anschließend wieder zusammengeführt:

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O \tag{2.5}$$

mit

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{2.6}$$

Dabei sind $W_i^Q$, $W_i^K$ und $W_i^V$ Projektionen für den $i$-ten Head und $W^O$ ist die Ausgangsprojektion, die die konkatenierte Head-Ausgabe wieder auf die Modelldimension abbildet. Durch die Multi-Head Attention kann das Modell gleichzeitig in verschiedenen semantischen Räumen Muster lernen. Beispielsweise könnte ein Head vor allem syntaktische Abhängigkeiten fokussieren, ein anderer hingegen thematische Zusammenhänge analysieren. In einem Encoder-Self-Attention-Layer stammen $Q$, $V$ und $K$ aus demselben vorherigen Encoder-Layer. Dies ist bei der Decoder-Self-Attention ähnlich, jedoch wird dort eine Maskierung angewendet, die alle zukünftigen Projektionen unzugänglich macht. Dadurch wird die Autoregressivität gewährleistet, da ein Token nur auf sich selbst und frühere Tokens zugreifen darf. Die Encoder-Decoder-Attention im Decoder-Stack beschreibt, dass die Queries aus dem aktuellen Decoder-Layer stammen, während die Keys und Values aus der Encoder-Ausgabe kommen. Dadurch wird jedem Decoder-Token ermöglicht, relevante Informationen aus allen Encoder-Tokens zu beziehen. [VASW17]

Im Anschluss an die (Multi-Head) Attention-Schicht folgt pro Layer ein FFN, das auf jede Position angewandt wird. Typischerweise besteht dieses FFN aus zwei linearen Transformationen, zwischen denen eine nichtlineare Aktivierung geschaltet ist:

$$FFN(x) = \max(0, \mathrm{x}W_1 + \mathrm{b1})\, W_2 + b_2 \tag{2.7}$$

Dabei sind $W_1$ und $W_2$ die Gewichtsmatrizen sowie $b_1$ und $b_2$ die Bias-Vektoren. Das FFN erweitert dadurch die Modellkapazität, indem pro Token zusätzliche nichtlineare Merkmalskombinationen erlernt werden. [VASW17]

Bevor die erste Transformer-Schicht beginnt, werden die Tokens durch eine Embedding-Schicht in d-dimensionale Vektoren umgewandelt. Außerdem wird an jede Token-Position ein Positionsvektor addiert, um die Wortstellung in der Sequenz kodiert mitzugeben. Diese Information ist notwendig, da das Modell weder Rekurrenz noch Konvolution verwendet und daher

die Positionen explizit ergänzt werden müssen. Vaswani et al. nutzten dafür feste sinus- und cosinusbasierte Positional Encodings, bei denen jede Dimension des Vektors einer Schwingung mit unterschiedlicher Wellenlänge entspricht. Auf der Ausgabeseite besitzt die Architektur einen Language-Model-Head. Dieser besteht aus einer Unembedding-Matrix und einer Softmax-Funktion. Für jeden letzten Hidden-State-Vektor, der eine interne Repräsentation eines Tokens darstellt, die aus der letzten Transformer-Schicht stammt, wird ein Logits-Vektor berechnet. Dieser Vektor hat die gleiche Länge wie das Vokabular, und die Softmax-Funktion wandelt diesen in Wahrscheinlichkeiten um. Damit sagt das Modell für jede Position voraus, welches Token als Nächstes im Text folgt. [VASW17] [JURA25e]

Durch die Transformer-Architektur können weitreichende Abhängigkeiten in Texten modelliert werden, da jedes Token direkten Zugang zu allen anderen Tokens im Kontext erhält. Transformer-Encoder liefern kontextuelle Repräsentationen von Texteingaben, die für vielfältige Sprachverarbeitungsaufgaben genutzt werden können, wohingegen Transformer-Decoder die leistungsstarke Generierung von Textsequenzen ermöglichen. [VASW17] [JURA25e]

### 2.2.2 Prompt-Engineering

Prompt-Engineering beschreibt das gezielte Formulieren von Eingabeaufforderungen (Prompts), um LLMs in Richtung der gewünschten Ausgabe zu lenken. Dabei wird die Eingabe so gestaltet, dass das Modell den notwendigen Kontext, klare Anweisungen und Beispiele erhält, damit die Generierungsanfrage vom Nutzer optimal bearbeitet werden kann. Im Gegensatz zum aufwändigen Fine-Tuning von LLMs werden die Modellparameter durch Prompt-Engineering weder verändert noch das Wissen erweitert. Prompt-Engineering nutzt vielmehr das vorhandene Wissen optimal aus. [GOOG25]

Um die Antworten der LLMs zu beeinflussen, existieren verschiedene Prompting-Methoden. Durch Chain-of-Thought (CoT) Prompting wird das Modell angeleitet, komplexe Probleme in logische Zwischenschritte zu zerlegen und den Lösungsweg Schritt für Schritt zu formulieren. Diese Technik fördert mehrstufiges Reasoning und kann die Genauigkeit bei Aufgaben mit mehreren Denkschritten deutlich erhöhen. Eine weitere Methode beschreibt das One-/Few-Shot Prompting. Dabei werden dem Modell ein oder mehrere Beispiele von Eingaben mitsamt den gewünschten Ausgaben im Prompt mitgegeben, bevor die eigentliche Generierungsanfrage gestellt wird. Auf Basis der mitgegebenen Beispiele erkennt das LLM das gewünschte Format oder Lösungsverfahren und kann die Generierungsanfrage durch Imitation der Muster präziser bearbeiten. [GADE25] [GOOG25]

Im Vergleich zum Fine-Tuning bietet Prompt-Engineering einige Vorteile. Prompt-Engineering erfordert kein zeitintensives Fine-Tuning auf neuen Daten und keinen Eingriff in die Modellgewichte. Dadurch kann mit deutlich weniger Aufwand an Zeit und Ressourcen eine Steigerung der Ausgabequalität erreicht werden. Verbesserungen und Anpassungen lassen sich mit geringem Aufwand durch das Formulieren von Prompts umsetzen und können iterativ verbessert werden, bis die gewünschte Ausgabe erreicht wird. [BELC25] [ZARE25]

Da Prompts nicht direkt die Wissensbasis der LLMs erweitern, können LLMs nur das leisten, was während des ursprünglichen Trainings erlernt wurde. Zudem ist das Kontextfenster von LLMs limitiert, weshalb Prompts nicht beliebig erweitert und nicht das gesamte verfügbare Wissen integriert werden kann. Für sehr spezialisierte oder branchenspezifische Aufgaben reicht reines Prompt-Engineering daher oft nicht aus, da den LLMs die nötigen Kenntnisse und Fähigkeiten fehlen, um die Anforderungen vollständig zu erfüllen. In Bezug auf Programmcode fehlt häufig das Verständnis domänenspezifischer Syntax und firmeneigener Bibliotheken, da LLMs nicht darauf trainiert wurden. Daher ist meist gezieltes Fine-Tuning mit relevanten Beispieldaten erforderlich, um die gewünschte Genauigkeit zu erreichen. [BELC25] [ZARE25]

## 2.3 Fine-Tuning von Large Language Models

Fine-Tuning beschreibt die gezielte Anpassung eines vortrainierten LLMs an eine spezifische Aufgabe. Während das allgemeine Training von LLMs auf sehr großen Textkorpora ein umfassendes Sprachverständnis vermittelt, ist das Fine-Tuning darauf ausgerichtet, die Modellparameter in einem nachgelagerten Fine-Tuning-Schritt zu modifizieren. Dadurch erlernen LLMs spezialisiertes Wissen und funktionieren in bestimmten Anwendungskontexten zuverlässiger. [HU21]

Die Motivation für Fine-Tuning ergibt sich aus den Grenzen des Prompt-Engineering (vgl. Kapitel 2.2.2). Durch die geschickte Gestaltung von Prompts kann eine Vielzahl von Aufgaben bewältigt werden, jedoch stößt dieser Ansatz in hochspezialisierten technischen Anwendungen an Grenzen. LLMs verfügen in der Regel nicht über ausreichende Kenntnisse domänenspezifischer Syntax oder Bibliotheken, welche bei der Generierung von effizientem Programmcode erforderlich sind. Das führt bei der Generierung von Programmcode häufig zu syntaktischen Fehlern und zu Halluzinationen, bei denen Funktionen erzeugt werden, die in der Zielumgebung nicht existieren. Fine-Tuning ermöglicht hingegen, vortrainierte LLMs mit synthetischen oder realen Beispielen anzupassen und damit spezifisches Fachwissen dauerhaft in den Modellparametern zu speichern. [HU21]

Das klassische Fine-Tuning, bei dem alle Modellparameter angepasst werden, ist bei modernen LLMs mit mehreren Milliarden Parametern oft nicht praktikabel. Der Ansatz erfordert sehr hohe Rechenressourcen, großen Speicherbedarf und lange Trainingszeiten. Um diesen Aufwand zu reduzieren, existieren verschiedene Methoden des Parameter-Efficient Fine-Tuning (PEFT) [HU21]

Die Grundidee von PEFT besteht darin, den Großteil der Modellgewichte eingefroren zu lassen und nur einen kleinen, zusätzlich eingefügten Teil der Parameter zu trainieren. Dafür werden Adaptermodule in bestimmten Schichten eingefügt oder gezielte Parametergruppen angepasst, während die übrigen Gewichte unverändert bleiben. Das reduziert die Zahl der trainierbaren Parameter erheblich und verkürzt die Trainingszeit. Die Vorteile von PEFT liegen daher in der Ressourceneffizienz und der Modularität. Adapter und Zusatzmodule können separat gespeichert werden, sodass mehrere domänenspezifische Anpassungen für ein Basismodell existieren können, ohne jedes Mal ein vollständiges Modell abspeichern zu müssen. Herausforderungen bei PEFT bestehen darin, dass nur ein Teil der Modelle angepasst wird. Insbesondere für Aufgaben, die weit außerhalb des ursprünglich gelernten Sprachraums liegen, kann PEFT an Grenzen stoßen. Daher ist die Auswahl des Basismodells von großer Bedeutung, damit dieses optimal an den speziellen Anwendungsfall angepasst werden kann. [HU21]

Eine etablierte PEFT-Methode ist die Low-Rank Adaption (LoRA). Die Abbildung 2.2 verdeutlicht das Prinzip, um LLMs mit der Methode anzupassen.

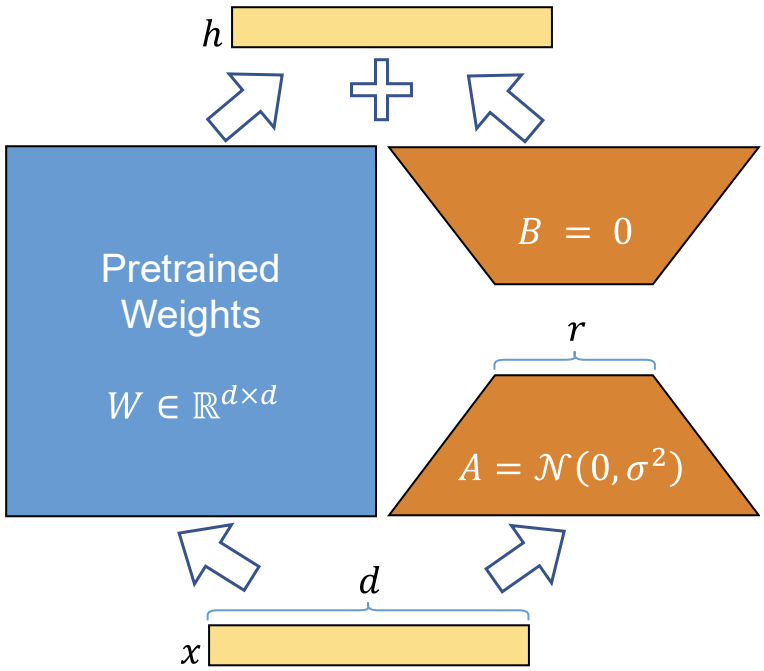


Abbildung 2.2: Low-Rank Adaption [HU21]

Anstatt die großen Gewichtsmatrizen von LLMs vollständig zu aktualisieren, wird bei LoRA die Gewichtsanpassung $\Delta W$ durch das Produkt zweier kleinerer Matrizen approximiert. Konkret bleibt die ursprüngliche Gewichtsmatrix $W$ eingefroren, und zwei neue Matrizen $A$ und $B$ mit niedrigerem Rang ($r << d$) werden trainiert. Dadurch ergibt sich die effektive Gewichtsmatrix nach LoRA zu:

$$W' = W + \Delta W = W + A * B \tag{2.8}$$

Die endgültige Berechnung des Ausgabevektors $h$ erfolgt durch:

$$h = xW' = x(W + A * B) = xW + xAB \tag{2.9}$$

Der erste Term $xW$ beschreibt die ursprüngliche Modelltransformation, die während des Fine-Tunings eingefroren ist, und der zweite Term $xAB$ repräsentiert die zusätzliche, trainierte Low-Rank-Korrektur. Die Summe beider Terme ergibt somit den angepassten Ausgabevektor $h$, der sowohl das ursprüngliche Modellwissen als auch die neu gelernten domänenspezifischen Anpassungen enthält. Auf diese Weise können Milliarden von Parametern unverändert bleiben, während lediglich wenige Millionen Parameter in den Low-Rank-Matrizen optimiert werden. Dadurch lassen sich erhebliche Einsparungen beim Speicher- und Rechenaufwand erzielen, ohne die Flexibilität der Modelle wesentlich einzuschränken. In der im Anhang aufgeführten Tabelle A 1 sind die zentralen konfigurierbaren Parameter für das Fine-Tuning dargestellt, die für eine optimale Anpassung des Modells an die Zieldomäne festgelegt werden können. [HU21]

## 2.4 Compiler-Technologien

Ein Compiler beschreibt ein Programm, das einen in einer Programmiersprache geschriebenen Quelltext in ein ausführbares Programm übersetzt [MEYE21, S. 15]. Dabei wird der Eingabecode in eine semantisch äquivalente Form überführt, wobei die Bedeutung des Programms erhalten bleibt. Um festzustellen, ob ein Programm zur definierten Sprache gehört, werden formale Grammatiken verwendet, die die Syntax der Sprache eindeutig festlegen. Allgemein lässt sich die Arbeitsweise in eine Analysephase und eine Synthesephase unterteilen. Die Analysephase umfasst die lexikalische Analyse, die syntaktische Analyse und die semantische Analyse, während die Synthesephase die Schritte Zwischencodeerzeugung, Programmoptimierung und Codeerzeugung beinhaltet. [ZÜHL11]

In der ersten Phase des Compilers wird der Quelltext zeichenweise gelesen und in eine Folge von Tokens umgewandelt. Ein Token entspricht beispielsweise einem Schlüsselwort, Bezeichner oder Operator der Sprache. Zeichen, die keine direkte Bedeutung für die Umwandlung besitzen, wie Leerzeichen und Kommentare, werden üblicherweise ignoriert. Die lexikalische Analyse wird von einem Scanner durchgeführt und liefert die Token-Sequenz als Ausgabe für die nachfolgende Syntaxanalyse. [ZÜHL11]

Die syntaktische Analyse (Parsing) überprüft die Struktur des Programmcodes anhand der Grammatik der Programmiersprache. Der Parser versucht, aus der Token-Sequenz einen gültigen Satz der formalen Grammatik zu bilden. Wenn das gelingt, konstruiert der Parser einen Syntaxbaum, der die strukturierte Darstellung des Programms enthält. Dieser Baum wird oft als abstrakter Syntaxbaum (Abstract Syntax Tree, AST) bezeichnet und repräsentiert die syntaktischen Einheiten des Quellprogramms. Im Falle von Regelverletzungen im Programmcode meldet der Parser die Syntaxfehler und bricht den Übersetzungsprozess ab. Der Parser stellt daher sicher, dass der Code in Bezug auf die Grammatikregeln korrekt aufgebaut ist. [ZÜHL11]

Durch die semantische Analyse werden die statischen Semantiken bzw. Kontextbedingungen der Sprache überprüft. In dieser Phase sammelt der Compiler Informationen über Bedeutungen, Typen und Bezeichner des Programms, um die Gültigkeit der Anweisungen zu überprüfen. Beispielsweise muss jede verwendete Variable deklariert worden sein, bevor auf diese zugegriffen wird, und die Datentypen der Operanden müssen zu den jeweiligen Operationen passen. Solche semantischen Prüfungen gehen über die reine Syntax hinaus und stellen sicher, dass kein Verstoß gegen die Sprachregeln vorliegt. Sind alle semantischen Prüfungen erfolgreich, gilt der Quelltext als korrekt. Anschließend kann der Compiler in der Synthesephase den geprüften Code in die Zielsprache übersetzen. [ZÜHL11]

Der Compiler erzeugt nach der semantischen Analyse in einer Zwischenschicht einen Zwischencode, der zwischen der quell- und zielsprachlichen Darstellung liegt. Dieser Zwischencode ist meist unabhängig von der konkreten Zielmaschine und dient als standardisierte interne Repräsentation. Diese bildet zudem die Basis für nachfolgende Optimierungen. [ZÜHL11]

Die Optimierungsphase zielt darauf ab, den Zwischencode hinsichtlich Laufzeit oder Speicherverbrauch zu verbessern, ohne die Semantik des Programms zu verändern. Dabei wird zwi-

schen maschinenunabhängiger Optimierung, beispielsweise der Eliminierung redundanter Berechnungen, und maschinenabhängiger Optimierung, beispielsweise der Registerallokation, unterschieden. [ZÜHL11]

In der abschließenden Codeerzeugung wird der optimierte Zwischencode in Maschinencode für die konkrete Zielplattform übersetzt. In dieser Phase werden Instruktionen erzeugt, Speicheradressen zugewiesen und gegebenenfalls zielsystemspezifische Register belegt. Dabei wird korrekter und effizienter Maschinencode erstellt, der das Verhalten des ursprünglichen Quellprogramms exakt widerspiegelt. Erst durch alle Schritte kann korrekter sowie performanter Maschinencode generiert werden. [ZÜHL11]

## 2.5 Synthetische Trainingsdaten

LLMs benötigen große Mengen hochwertiger Daten, doch in vielen spezialisierten Bereichen sind reale Daten nur begrenzt verfügbar. Echte Daten sind oft nicht ausreichend vorhanden, teuer und unterliegen strengen Datenschutzauflagen. In der industriellen Automatisierung enthalten Maschinendaten häufig sensible, unternehmenskritische Informationen, die nicht ohne Weiteres extern geteilt oder in LLMs trainiert werden können. Dadurch fehlen in solchen Domänen häufig öffentlich zugängliche Trainingsdatensätze. Während für Bereiche wie beispielsweise Open-Source-Code in Programmiersprachen wie Python umfangreiche Datenbestände frei verfügbar sind, existieren für branchenspezifische Programmiersprachen wie etwa die SPS-Programmiersprache ST so gut wie keine öffentlichen Trainingsdaten. Dieses Datensatzdefizit erschwert die Entwicklung und das zuverlässige Fine-Tuning von LLMs in spezialisierten Anwendungsfeldern. [NIKO21, S.12; 217-221] [GRAM25] [KRÜG25]

Synthetisch generierte Trainingsdaten rücken vermehrt in den Fokus. Unter synthetischen Daten werden künstlich erzeugte Datensätze verstanden, die statistische Eigenschaften realer Daten nachahmen, ohne tatsächliche sensible Informationen zu enthalten. Durch synthetische Daten können Defizite in den Trainingsdaten gefüllt und Szenarien abgedeckt werden, die in der Realität nur schwer erfassbar sind. Zudem werden gleichzeitig vertrauliche Informationen geschont und Datenschutzauflagen eingehalten. Unternehmen und Forschungseinrichtungen setzen vermehrt auf diesen Ansatz, um der Datenknappheit zu begegnen. Nach Einschätzung des Marktforschungsunternehmens Gartner liegt der Anteil synthetisch generierter Trainingsdaten mittlerweile bei über 60 % [MORR23]. Synthetische Trainingsdaten gelten daher zunehmend als Schlüssel, um Datenknappheit und Datenschutzprobleme beim LLM-Training zu lösen und dennoch robuste, leistungsfähige LLMs zu entwickeln. [GRAM25] [KRÜG25]

Es gibt verschiedene methodische Ansätze, um synthetische Trainingsdaten zu erzeugen. Diese lassen sich im Wesentlichen in die drei Kategorien regelbasiert, augmentativ und generativ einordnen. Beim regelbasierten Ansatz werden Daten nach vordefinierten Regeln, Logiken oder Simulationen generiert. Auf Basis von Modellierungsregeln können realistische Werte erzeugt werden. Dieser Ansatz ist nachvollziehbar und kontrollierbar und eignet sich insbesondere, wenn die Datenstruktur und die Regeln klar definiert sind. Augmentative Generierung hingegen beschreibt das gezielte Variieren von bestehenden Daten. Verfahren wie Rauschen, Transformationen bei Bildern oder Umformulieren bei Text generieren verschiedene Varianten vorhandener Trainingsdaten, um neue synthetische Trainingsdaten zu erstellen. Generative Verfahren nutzen Modelle der künstlichen Intelligenz selbst, um neue Daten zu erzeugen. Besonders relevant ist hier der Einsatz von LLMs, die in der Lage sind, auch komplex strukturierte Daten wie Programmcode zu generieren. Diese bieten die Möglichkeit, durch die Erstellung von synthetischem Programmcode ein umfangreiches Fine-Tuning von LLMs zu ermöglichen. LLMs liefern Daten, die echten Beispielen sehr nahekommen, erfordern jedoch eine Validierung, um Fehler oder Verzerrungen aus den Trainingsdaten nicht unkontrolliert zu übernehmen und in die LLMs zu trainieren. [NIKO21, S.12; 219-221; 278] [MURE24] [PASI22]

# 3 Problemanalyse und Anforderungen an die Lösung

Die Nutzung von LLMs im industriellen Umfeld eröffnet neue Potenziale, insbesondere für die Unterstützung bei der Programmierung in ST. Gleichzeitig treten erhebliche Einschränkungen auf, die sich aus der Verfügbarkeit geeigneter Daten, technischen Limitationen und fehlenden methodischen Standards ergeben. Dieses Kapitel untersucht die bestehenden Herausforderungen beim Einsatz und Fine-Tuning von LLMs im ST-Kontext und leitet daraus konkrete Anforderungen ab. Das Kapitel 3.1 beschreibt die zentralen Herausforderungen, und in Kapitel 3.2 werden daraus die Anforderungen an das Tool abgeleitet und in Form einer Anforderungsliste strukturiert dargestellt.

## 3.1 Problemanalyse

Zurzeit wird für die Steigerung der Ausgabequalität von LLMs überwiegend auf Prompt-Engineering zurückgegriffen. Hierbei wird versucht, den relevanten Kontext, wie den Aufbau von Funktionsblöcken oder Funktionen, in den Prompt einzubinden, um ein LLM zur Generierung von ST-Code zu leiten. Durch Methoden des Prompt-Engineerings werden benutzerspezifische Vorgaben wie Syntaxregeln, Namenskonventionen oder Programmierparadigmen berücksichtigt. Dieses Verfahren stößt jedoch bei zunehmender Komplexität an die Grenzen der verfügbaren Kontextfenster. Die *Tc2_MC2*-Bibliothek zur effizienten Ansteuerung von Motoren enthält beispielsweise 62 verschiedene Datentypen und dazu weitere 66 Funktionsblöcke [BECK25i]. Damit wird deutlich, dass die vollständige Einbettung solcher Funktionsdefinitionen in den Prompt den verfügbaren Kontext schnell übersteigt. Ein ausschließlich auf Prompting basierender Ansatz ist nicht in der Lage, den vollen Umfang der in TwinCAT enthaltenen Bibliotheken und Funktionen abzubilden, sodass die Generierung durch das Modell unvollständig bleibt oder fehlerhafte Ergebnisse produziert.

Die größte Hürde beim Fine-Tuning von LLMs liegt in der Verfügbarkeit von ST-Trainingsdaten. Für ST existieren nahezu keine öffentlichen Datensätze, und Kundendaten aus der Anlagentechnik des betrachteten Unternehmens sind aus Datenschutzgründen nicht nutzbar. Selbst wenn reale Projektdaten verfügbar wären, könnten diese Qualitätsprobleme aufweisen. Dadurch wäre eine aufwendige Vorverarbeitung notwendig, um redundanten, fehlerhaften oder stark unternehmensspezifischen Programmcode zu identifizieren. Hinzu kommt, dass ST-Projekte sehr heterogen aufgebaut sind und sich hinsichtlich Struktur, Bibliotheksnutzung und Programmierstil erheblich unterscheiden. Damit fehlt eine einheitliche Datenbasis, die für

ein robustes Fine-Tuning notwendig wäre. Gleichzeitig besteht zunehmend Bedarf an kundenspezifischen Trainingsdaten, um spezialisierte LLMs zu trainieren.

Um diese Lücke zu schließen, ist ein strukturierter Prozess zur Erzeugung und Aufbereitung der Trainingsdaten erforderlich. Ein solcher Prozess müsste nicht nur synthetische Daten generieren, sondern auch bestehende Kundenprojekte automatisiert einlesen, syntaktisch und semantisch analysieren sowie standardisiert und anonymisiert aufbereiten. Gleichzeitig sollten die generierten und extrahierten Daten bereits während der Erzeugung einer syntaktischen Prüfung unterzogen werden. Auf diese Weise ließen sich konsistente, qualitativ hochwertige und im Bedarfsfall auch kundenspezifisch zugeschnittene Datensätze aufbauen, die sich für ein robustes und effizientes Fine-Tuning nutzen lassen.

Auch der eigentliche Fine-Tuning-Prozess ist derzeit nicht standardisiert. In der Praxis wird meist mit selbst geschriebenen Python-Skripten experimentiert. Dadurch sind die erzielten Ergebnisse schwer vergleichbar und kaum reproduzierbar. Für eine langfristige Strategie für das Fine-Tuning von LLMs wäre es notwendig, einen klar definierten und reproduzierbaren Prozess für das Fine-Tuning zu entwickeln, der mit Trainingsdaten konsistent arbeitet.

Um die allgemeine Eignung von LLMs und die Qualität nach dem Fine-Tuning-Prozess festzustellen, fehlt es an geeigneten Tests. Während es für viele Programmiersprachen etablierte Benchmarks wie SWE-Bench gibt, lassen sich diese aufgrund der Spezialisierung von ST nicht einfach übertragen [JIME25]. Ohne standardisierte Testfälle bleibt unklar, ob ein LLM zuverlässig arbeitet oder nur auf bestimmte Anwendungsfälle optimiert ist. Parser können zwar syntaktische Korrektheit prüfen, die semantische Validierung erfordert jedoch spezialisierte Testfälle. Daher ist die Erstellung diverser Testfälle notwendig, um Fortschritte beim Fine-Tuning messbar zu machen und die Eignung von LLMs evaluieren zu können.

## 3.2 Anforderungen an die Lösung

Auf Grundlage der durchgeführten Problemanalyse lassen sich die Anforderungen an die Lösung ableiten. Die Anforderungen werden durch eine eindeutige Identifikationsnummer gekennzeichnet, um sie im Verlauf des Projekts erneut aufgreifen und während der Konzeptentwicklung referenzieren zu können (siehe Tabelle 3.1).

| **Identifikations-nummer** | **Anforderung** |
|---|---|
| **A1** | TwinCAT-Projekte sollen automatisiert eingelesen und auf Strukturmerkmale, Richtlinien und Designprinzipien analysiert werden. |

| | |
|---|---|
| **A2** | Die aus den TwinCAT-Projekten gewonnenen Analysedaten sollen flexibel angepasst werden können, um die Generierung synthetischer Trainingsdaten gezielt zu steuern. |
| **A3** | Es soll eine definierte Anzahl an Trainingsdaten synthetisch generiert und dabei die projektspezifischen Anforderungen berücksichtigt werden. |
| **A4** | Ein Parser soll im Generierungsprozess syntaktische Prüfungen übernehmen und durch Feedbackschleifen die Qualität der generierten Trainingsdaten kontinuierlich steigern. |
| **A5** | Als ressourcenschonendes Verfahren soll LoRA das Fine-Tuning hochparametrisierter LLMs ermöglichen. |
| **A6** | Für den Fine-Tuning-Prozess soll die Möglichkeit bestehen, unterschiedliche Datensätze sowie LLMs auszuwählen.. |
| **A7** | Die Qualität der LLMs soll syntaktisch durch einen Parser und semantisch durch Unit-Tests evaluiert werden. |
| **A8** | Nach dem Training sollen die LLMs für Testinferenz in verschiedenen Formaten sowie zur zentralen Verwaltung bereitgestellt werden. |

Tabelle 3.1: Definition der Anforderungen an die Lösung

Die Anforderungen **A1** und **A2** schaffen zunächst die Grundlage, TwinCAT-Projekte automatisiert zu erfassen und die Strukturen systematisch auszuwerten. Das ist entscheidend, um die Syntax- und Semantikaspekte von ST-Code umfassend analysieren zu können. Die Möglichkeit, die dabei entstandenen Analysedaten flexibel anzupassen, stellt sicher, dass die Generierung synthetischer Trainingsdaten gezielt gesteuert werden kann.

Den Fokus auf die Erzeugung und Qualitätssicherung der Trainingsdaten legen die Anforderungen **A3** und **A4**. Dabei beschreibt **A3**, dass Nutzer die Möglichkeit erhalten, den Umfang der Trainingsdaten zu definieren und auch eine Auswahl zu treffen, welche der projektspezifischen Anforderungen bei der Generierung berücksichtigt werden sollen. Durch **A4** wird die Integration eines Parsers beschrieben, der syntaktische Prüfungen während des Generierungsprozesses vornimmt. Durch Feedbackschleifen im Falle von Syntaxfehlern im synthetischen Programmcode wird eine kontinuierliche Qualitätssteigerung ermöglicht.

Mit den Anforderungen **A5** und **A6** wird der Fine-Tuning-Prozess im Tool adressiert. Dabei wird LoRA als ressourcenschonendes Verfahren gewählt, um auch mit begrenzten Rechenkapazitäten hochparametrisierte LLMs einem Fine-Tuning unterziehen zu können. Die Möglichkeit, verschiedene Datensätze und LLMs flexibel auszuwählen, schafft darüber hinaus die Voraussetzung, spezialisierte LLMs zu trainieren und diese je nach Datengrundlage optimal an den jeweiligen Anwendungsfall anzupassen.

Abschließend zielen die Anforderungen **A7** und **A8** auf die Sicherstellung der Modellqualität sowie deren Anwendbarkeit in der Praxis ab. **A7** stellt sicher, dass die Qualität der trainierten LLMs sowohl syntaktisch durch Parser als auch semantisch durch Unit-Tests geprüft wird. Damit wird gewährleistet, dass die LLMs nicht nur formal korrekten, sondern auch funktional sinnvollen Programmcode erzeugen können. **A8** hebt die Bereitstellung der trainierten LLMs hervor, sodass diese für Testinferenz unmittelbar genutzt werden können. Zudem wird dadurch die zentrale Verwaltung der LLMs und der Export in verschiedene Formate gewährleistet. Damit wird der Übergang von der Entwicklung der LLMs zur praktischen Anwendung unterstützt.

# 4 Stand der Technik

In diesem Kapitel wird der aktuelle Stand der Technik im Kontext der Nutzung von LLMs für Programmcode vorgestellt. Dazu werden zunächst in Abschnitt 4.1 relevante Methoden aus Forschung und Praxis beschrieben. Im Mittelpunkt stehen Verfahren zur Analyse von Softwareprojekten durch LLMs, Fine-Tuning-Strategien sowie Ansätze zur Generierung und Bewertung von Code. Anschließend erfolgt in Abschnitt 4.2 die Bewertung der vorgestellten Ansätze hinsichtlich der Anwendbarkeit, Vorteile und Grenzen für die Entwicklung des Tools.

## 4.1 Vergleichbare Lösungsansätze

LLMs werden erforscht als Methode zur automatisierten Code-Analyse, etwa um Strukturmerkmale, Architektur-Patterns oder verwendete Richtlinien in Programmcode-Projekten zu erkennen. Ein Beispiel ist die LLM-basierte Design-Pattern-Erkennung, bei der ein LLM genutzt wird, um in Code Entwurfsmuster zu identifizieren. Solche Ansätze zeigen grundsätzlich, dass LLMs in der Lage sind, grundlegende semantische Muster im Code zu erkennen. Allerdings bestehen auch Einschränkungen. Eine empirische Untersuchung zeigt beispielsweise, dass gängige LLMs Schwierigkeiten haben, bestehende Architekturprinzipien zu erfassen, sofern die zu analysierenden Aspekte nicht hinreichend konkretisiert sind. LLMs sind insgesamt durchaus fähig, Codezusammenfassungen und Architekturprinzipien zu analysieren, sofern die Vorgehensweise der Analyse klar definiert ist. Fehlt eine solche Präzisierung, neigen sie jedoch dazu, Projektkontexte und Prinzipien zu vernachlässigen. [SCHI25] [ZHEN25]

Da hochwertige Datensätze für das Training von LLMs rar und kostenintensiv in der Erstellung sind, greifen viele aktuelle Ansätze auf synthetische Datengenerierung zurück. Dabei nutzen hochparametrisierte LLMs automatisierte Verfahren zur Erzeugung neuer Trainingsdaten. Ein prominentes Beispiel stellt Code Alpaca dar. Der Datensatz umfasst 20.000 Frage-Antwort-Paare zu Programmierproblemen. Für die Generierung erstellt ein hochparametrisiertes Modell (ChatGPT) verschiedene Programmieraufgaben inklusive passender Lösungen. Der synthetische Frage-Antwort-Datensatz dient anschließend dem Fine-Tuning kleinerer LLMs, um einen Codeassistenten bereitzustellen. Aufbauend darauf verfeinern weitere Projekte die Methode. WizardCoder beispielsweise erhöht die Komplexität und Diversität der generierten Aufgaben durch einen evolutionären Ansatz, bei dem ChatGPT iterativ Aufgaben schrittweise schwieriger und variantenreicher gestaltet, um einen umfangreichen Trainingsdatensatz zu erzeugen. [NADA25]

Bei der Generierung synthetischer Trainingsdaten gibt es jedoch wichtige Aspekte zu beachten, da die Datenqualität und -vielfalt entscheidend für die Qualität der LLMs nach dem Fine-Tuning ist. Daher sind Mechanismen notwendig, um fehlerhaften Output zu identifizieren und dann entsprechend weiterzuverarbeiten oder zu entfernen. Moderne Ansätze kombinieren die Datenerzeugung oft mit automatischen Überprüfungen. Der Datensatz OpenCode-Instruct von NVIDIA enthält beispielsweise zu jedem synthetischen Beispiel auch Ausführungs-Feedback und eine Qualitätsbewertung. Die Trainingsdaten werden also getestet und beurteilt, bevor diese beim Fine-Tuning verwendet werden. Solche Feedback-Schritte stellen sicher, dass die synthetisch erzeugten Trainingsdaten syntaktisch korrekt sind und der Programmiersprache entsprechen. Zudem sollte die Diversität der generierten Daten hoch sein, indem verschiedene Aufgabentypen, Schwierigkeitsgrade und Domänen trainiert werden. [AHMA25] [NADA25]

Ein weiterer vielversprechender Ansatz ist die Einbindung von Feedback-Schleifen während der Codeerzeugung. Hierbei wird der Output des LLMs kontinuierlich überprüft, indem dieser durch Kompilieren, Parser oder Testen bewertet wird. Die Rückmeldungen zu möglichen Fehlern fließen automatisiert wieder in das LLM ein, um den Code schrittweise zu verbessern. Das Konzept „LLM in the Loop“ wird aktuell intensiv erforscht, da LLMs keine Garantie auf fehlerfreien Code geben. So schlagen Ravi et al. mit dem LLMLOOP-Framework fünf iterative Schleifen vor, in denen der generierte Code zunächst auf Kompilierbarkeit geprüft und syntaktische Fehler behoben werden, dann statische Analyse-Warnungen ausgebessert, anschließend gegen Tests ausgeführt und bei Fehlschlägen repariert wird. Dieser vollständig automatische Zyklus zeigt in Experimenten positive Ergebnisse. Auf dem Benchmark HumanEval-X kann LLMLOOP die Erfolgsrate von rund 76 % auf über 90 % mit zehn Feedback-Schleifen steigern. Das zeigt, dass iteratives Verbessern durch Compiler die Qualität von LLM-Code deutlich erhöhen kann. [RAVI25]

Fine-Tuning zeigt sich als zentrales Verfahren zur Verbesserung von vortrainierten LLMs im Programmierumfeld. Durch das Fine-Tuning lernt ein allgemeines LLM die speziellen Syntaxregeln, Bibliotheken und Muster von Programmiersprachen und kann deutlich bessere Ergebnisse bei der Codegenerierung liefern. Beispielsweise können durch Fine-Tuning Open-Source-Modelle auf Programmieraufgaben angepasst werden. Eine Studie zeigt, dass durch das Fine-Tuning die Qualität eines allgemeinen StarCoder-Modells mit dem Code-Alpaca-Datensatz von 33,6 % auf 57,3 % gesteigert werden kann [ZIYA25]. Es wird allgemein hervorgehoben, dass eine Qualitätssteigerung nur dann erzielt werden kann, wenn die verwendeten Trainingsdaten inhaltlich mit dem spezifischen Anwendungsbereich der LLMs übereinstimmen. [AHMA25]

Als Fine-Tuning-Methode wird LoRA in diversen Studien hervorgehoben, um hochparametrisierte LLMs anzupassen. Die Ergebnisse, die durch das LoRA-Verfahren erreicht werden können, zeigen eine deutliche Steigerung der Ausgabequalität. Beispielsweise wird berichtet, dass durch LoRA angepasste LLMs bei der Verarbeitung von Finanzdaten im Durchschnitt eine Steigerung von 36 % Genauigkeit gegenüber dem Basismodell erzielen [WANG25]. Auch bei der Generierung von Programmcode in der Programmiersprache C kann die Qualität durch Fine-Tuning um 6,4 % gesteigert werden [LI24]. Ein weiterer Vorteil, der hervorgehoben wird, ist die Ressourceneffizienz des LoRA-Verfahrens. Mit dem Verfahren kann gegenüber herkömmlichem Fine-Tuning, bei dem alle Parameter angepasst werden, der Bedarf an VRAM für das Training um ca. 66 % reduziert und gleichzeitig die Geschwindigkeit des Trainingsprozesses um 50–70 % erhöht werden [HU21] [SAND25]. Das betont insbesondere die Eignung des LoRA-Verfahrens, wenn hochparametrisierte LLMs angepasst werden sollen. [RAUF24] [AHMA25]

Um die Leistung von LLMs im Software-Kontext zu bewerten, haben sich die syntaktische und semantische Korrektheit etabliert. Moderne Benchmarks wie HumanEval oder MDPP prüfen die syntaktische Korrektheit, indem diese dem Modell Programmieraufgaben geben und anschließend überprüfen, ob der Output ohne Syntaxfehler ist und alle vorgegebenen Testfälle besteht. Zur Bewertung der semantischen Qualität enthält der HumanEval-Benchmark Programmieraufgaben, wobei jede Aufgabe mit Unit-Tests zur Validierung verwendet wird. Durch diese Kombination können LLMs hinsichtlich der Leistung in Bezug auf Codegenerierung evaluiert werden. [RAVI25]

## 4.2 Bewertung bestehender Methoden und Tools

Für das geplante Tool zeigen die in 4.1 vorgestellten Ansätze sowohl die positiven Erkenntnisse als auch die Limitierungen, die bei der Umsetzung des Tools beachtet werden müssen. Die Analyse ermöglicht eine Einordnung, welche Ansätze zur Umsetzung beitragen können, welche Anpassungen notwendig sind und wo klare Grenzen der Anwendbarkeit bestehen.

Der Einsatz von LLMs zur Analyse von Projektcode im Hinblick auf Architektur, Codequalität und Designprinzipien eröffnet für das geplante Tool vielversprechende Potenziale. Insbesondere LLMs mit umfangreichem Codeverständnis bieten die Möglichkeit, den Aufbau von Code zu verstehen und die Analyse durchzuführen. Allerdings müssen auch die Limitationen berücksichtigt werden. LLMs können dazu neigen, ohne explizite Hinweise projektbezogene Konventionen zu ignorieren. Daher muss über Prompting exakt definiert werden, welche Aspekte

analysiert werden sollen, um die Analyse der LLMs gezielt anzuleiten, damit alle relevanten Aspekte berücksichtigt werden. Zudem ist die Kontextgröße ein herausfordernder Aspekt. Große Projekte übersteigen schnell den Kontext von LLMs. Für diese Aufgabe sind daher LLMs von Vorteil, die ein erhöhtes Kontextfenster besitzen, um umfangreichen Code analysieren zu können. Alternativ kann der Code in Ausschnitte aufgeteilt werden, um die Analyse nur auf Teilbereiche des Codes auszuführen und die Ergebnisse im Verlauf zu einem gemeinsamen Analyseergebnis zusammenzuführen.

Die Ansätze in Bezug auf synthetische Trainingsdaten zeigen, dass diese ein notwendiges Mittel sind, wenn echte Daten knapp sind. Für das Tool bedeutet das konkret, dass mit Hilfe eines leistungsstarken LLM synthetische Trainingsdaten generiert werden können. Durch die Generierung von Trainingsdaten im Frage-Antwort-Format können diverse Programmbeispiele erstellt werden, wodurch das Trainingsspektrum gezielt auf spezifische Anwendungen abgedeckt wird. Die Erkenntnisse aus der Forschung heben zudem den Einfluss der Qualität der Trainingsdaten hervor. Zum einen muss gewährleistet werden, dass die Trainingsdaten syntaktisch korrekt sind und keine Fehler beinhalten. Das kann durch die Integration von Feedbackschleifen und automatischen Korrekturen erreicht werden. Studien zeigen, dass durch iteratives Verbessern des generierten Codes die Qualität der Trainingsdaten gesichert werden kann. Zum anderen wird ausreichende Variabilität in den Trainingsdaten hervorgehoben, um eine gute Generalisierungsfähigkeit nach dem Fine-Tuning zu gewährleisten.

Die Literatur belegt, dass ein auf allgemeinem Text trainiertes LLM durch zusätzliches Fine-Tuning auf Code erheblich verbessert werden kann. Daher ist der Ansatz für das Tool sinnvoll, ein Fine-Tuning auf geeigneten Datensätzen vorzusehen, um die Syntax zu trainieren und bestimmtes Domänenwissen in die LLMs zu integrieren. LoRA bietet dabei einen ressourcenschonenden Ansatz, um auch größere LLMs mit praktikablem Rechen- und Zeitaufwand anpassen zu können.

Ein Großteil der Benchmark-Konzepte bietet Anknüpfungspunkte für die Umsetzung des Tools. Nach dem Fine-Tuning ist es notwendig, die angepassten LLMs auf Qualität zu testen, um die Eignung festzustellen. Dafür kann zum einen die syntaktische Qualität geprüft werden, um die Kompilierbarkeit des generierten Codes festzustellen. Zum anderen können umfangreiche Beispielszenarien in Form von Unit-Tests mit erwarteten Ausgaben definiert werden, um auch die semantische Qualität des Codes zu evaluieren. Durch die Kombination dieser Verfahren lässt sich die Wirksamkeit des Fine-Tunings systematisch erfassen und objektiv bewerten.

# 5 Lösungskonzept

Das Lösungskonzept gliedert den Gesamtprozess in vier aufeinander abgestimmte Teilbereiche, die gemeinsam die Anforderungen **A1** bis **A8** abdecken. Das Lösungskonzept ist so konzipiert, dass alle vier Teilbereiche sowohl als eigenständig einsetzbare Lösungen als auch in Kombination als durchgängiger, integrierter Gesamtprozess funktionieren. Diese werden in der grafischen Konzeption des Tools in Abbildung 5.1 dargestellt.

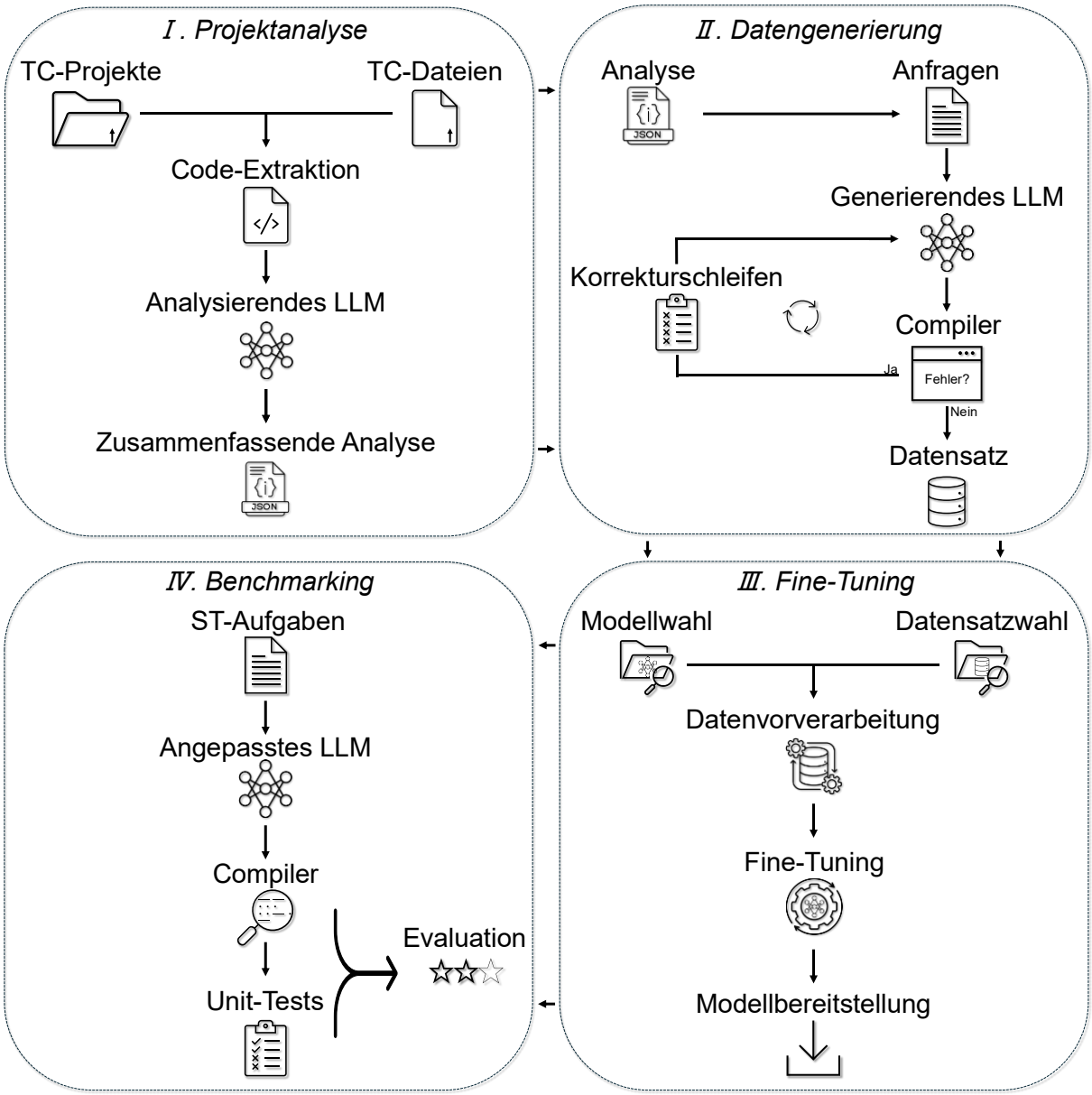


Abbildung 5.1: Grafische Konzeption vom Tool

### Projektanalyse

Im Rahmen des Lösungskonzepts erfolgt eine automatisierte semantische Analyse realer TwinCAT-Projekte und -Dateien, um strukturierte und weiterverwertbare Informationen über enthaltene Code-Elemente zu gewinnen. Ziel ist es, Programmstrukturen, Implementierungsrichtlinien und wiederkehrende Designprinzipien systematisch zu erfassen und aufzubereiten, um sie anschließend für die Generierung geeigneter Trainingsdaten und das Fine-Tuning von Large Language Models nutzbar zu machen.

Dazu besteht die Möglichkeit, vollständige TwinCAT-Projekte oder ausgewählte TwinCAT-Dateien in das Tool hochzuladen und kontrolliert in einen analysierbaren Zustand zu führen. Eine nachgelagerte Filterstufe extrahiert ausschließlich relevante Quelldokumente, wie beispielsweise TcPOU-Dateien, und separiert daraus den eigentlichen Programmcode. Dieser wird in einer einheitlichen Zwischenrepräsentation gespeichert, die sowohl den inhaltlichen Code als auch kontextbezogene Metadaten (z. B. Dateipfad, Abhängigkeiten) enthält. Dadurch lassen sich Informationsverluste vermeiden und die Daten für nachfolgende Verarbeitungsschritte konsistent bereitstellen.

Die eigentliche Auswertung übernimmt ein lokal ausgeführtes LLM mit erweitertem Kontextfenster. Um die durch Eingabelängen bedingten Größen- und Komplexitätsgrenzen zu umgehen, wird die Analyse konzeptionell in mehrere Teilschritte partitioniert. Der extrahierte Code wird in inhaltlich zusammenhängende Segmente zerlegt, separat analysiert und anschließend zu einer konsolidierten Gesamtanalyse zusammengeführt. Durch gezieltes Prompt-Engineering wird die Struktur und Tiefe der Analyse definiert, sodass die Ergebnisse in einer festgelegten Formatierung vorliegen.

Das resultierende Analyseergebnis (**A1**) enthält zentrale Struktur-, Qualitäts- und Entwurfseigenschaften der untersuchten TwinCAT-Projekte oder -Dateien. Diese können anschließend gezielt erweitert oder angepasst werden, um eine bedarfsgerechte Generierung optimaler Trainingsdaten zu ermöglichen (**A2**). Damit bildet der beschriebene Analyse- und Aufbereitungsprozess die Grundlage für die nachgelagerten Schritte der synthetischen Datengenerierung und des Fine-Tunings.

### Datengenerierung

Der Teilbereich II dient der kontrollierten Erzeugung synthetischer ST-Trainingsdaten und bildet den zentralen Übergang zwischen der inhaltlichen Analyse und dem anschließenden LLM-Fine-Tuning. Ziel ist es, auf Basis der zuvor gewonnenen Projektanalyse eine definierte Anzahl

qualitativ hochwertiger Trainingsdaten zu erzeugen, die analysierte Strukturen und Programmiermuster realitätsnah abbilden. Dabei wird sichergestellt, dass die generierten Beispiele stets im Anwendungskontext der Analyse stehen und somit domänenspezifische Charakteristika wie typische Funktionen oder Funktionsbausteine berücksichtigen (**A3**).

Die Generierung erfolgt durch ein leistungsfähiges, hochparametrisiertes LLM, das lokal oder über eine geeignete Schnittstelle betrieben wird. Auf Grundlage der Analyse entsteht hierbei eine Sammlung von Programmieranfragen, die das LLM gezielt generieren soll, um eine umfassende Abdeckung relevanter Anwendungsfälle sicherzustellen. Durch gezieltes Prompt-Engineering wird das Modell so angeleitet, dass die erzeugten Programme die in der Analyse identifizierten Strukturen und Richtlinien widerspiegeln. Die Prompts beinhalten typische Elemente der ST-Programmierung wie grundlegende Funktionsaufrufe, zyklische Bausteinstrukturen oder standardisierte Datentypen, die für die Erzeugung funktionsfähigen und logisch konsistenten Codes unerlässlich sind.

Ein unternehmensinterner Compiler (SW3) überprüft die generierten Ausgaben auf syntaktische Korrektheit (**A4**). Hierzu wird der eigentliche Programmcode zunächst aus der Modellantwort extrahiert, um eine saubere Validierung zu gewährleisten. Tritt in der Parsing-Phase ein Fehler auf, wird die Ausgabe verarbeitet und in einer definierten Anzahl an Korrekturschleifen erneut an das LLM übergeben. Das LLM wird dabei mithilfe präziser Fehlerrückmeldungen wie Position, Fehlertyp und möglichen Verbesserungen gezielt zur Optimierung des Codes angeleitet. Kann ein Fehler nach Durchlauf der Korrekturschleifen nicht behoben werden, erfolgt eine alternative, leicht modifizierte Generierung der Anfrage, um ein funktionsfähiges Ergebnis zu erhalten.

Nach erfolgreicher Validierung werden die synthetischen ST-Trainingsdaten um realistische Nutzeranfragen ergänzt, wodurch vollständige „Frage-Antwort“-Paare entstehen. Diese werden persistent gespeichert und bilden den finalen Trainingsdatenbestand für das anschließende Fine-Tuning der LLMs.

**Fine-Tuning**

Im Teilbereich III erfolgt das Fine-Tuning lokaler LLMs auf die zuvor erzeugten Trainingsdaten. Diese Phase stellt die zentrale Komponente zur anwendungsspezifischen Anpassung der LLMs dar. Für das Fine-Tuning kann zwischen verschiedenen lokal verfügbaren LLMs gewählt werden, die wahlweise bereits vorhanden oder direkt über das Tool heruntergeladen werden können. Zudem besteht die Möglichkeit, mehrere Datensätze gleichzeitig auszuwählen, um

ein LLM auf verschiedene Anwendungsbereiche zu trainieren oder domänenspezifische Aspekte gezielt zu kombinieren (**A6**).

Vor Beginn des Fine-Tunings werden die gewählten Datensätze automatisch vorverarbeitet und in das für das jeweilige LLM erforderliche Eingabeformat überführt. Da jedes LLM individuelle Anforderungen an Struktur, Tokenisierung und Sequenzlänge stellt, ist dieser Schritt vor dem eigentlichen Fine-Tuning entscheidend. Neben der Formatierung erfolgt in diesem Schritt die notwendige Tokenisierung, die der LLM-Architektur als Eingabe dient.

Für das eigentliche Fine-Tuning wird ein LoRA-basiertes Verfahren eingesetzt (**A5**). Dieses ermöglicht eine ressourcenschonende Anpassung von LLMs mit einer hohen Anzahl an Parametern, da nur ein kleiner Teil der ursprünglichen Modellparameter aktualisiert wird. Dadurch können auch umfangreiche Modelle lokal trainiert werden, ohne auf externe Anbieter zurückgreifen zu müssen. Die Trainingsparameter wie Rang, Lernrate oder Anzahl der Epochen werden in Abhängigkeit des gewählten LLMs automatisch identifiziert, können bei Bedarf aber manuell angepasst werden.

Nach Abschluss des Fine-Tunings werden die trainierten LoRA-Parameter mit den ursprünglichen LLM-Parametern zusammengeführt, wodurch ein vollständig angepasstes LLM entsteht. Dieses wird standardmäßig im SafeTensor-Format gespeichert, das eine sichere und effiziente Speicherung großer Tensoren gewährleistet [HUGG25a]. Optional kann das LLM in das GGUF-Format konvertiert werden, das ein kompaktes Speicherformat darstellt und für quantisierte LLMs optimiert ist [HUGG25b]. Die angepassten LLMs können anschließend in verschiedenen Modellformaten exportiert werden, wodurch eine Integration in nachgelagerte Anwendungssysteme gewährleistet ist (**A8**).

**Benchmarking**

Der Teilbereich IV sichert die objektive Bewertung der erzielten Ergebnisse. Die angepassten LLMs werden mit einem definierten Satz von etwa 100 bis 150 ST-Aufgaben getestet, die unterschiedliche Funktionsbereiche abdecken und jeweils mehrere Testfälle enthalten, um die Robustheit und Generalisierungsfähigkeit der LLMs zu prüfen (**A7**). So kann beispielsweise eine generierte Modulo-Funktion mit verschiedenen Wertekombinationen getestet werden, um die korrekte Funktionalität sicherzustellen.

Das zu testende LLM erhält zu Beginn der Bewertung eine klar definierte Aufgabenbeschreibung mit festgelegten Eingabeparametern. Ziel ist die Generierung einer ST-Funktion, die die

Aufgabe löst. Nach der Generierung wird der Programmcode extrahiert, da LLMs häufig erklärende Texte oder Kommentare außerhalb des Codeblocks erzeugen, welche eine Kompilierung verhindern könnten. Die extrahierte Funktion wird anschließend in ein vordefiniertes Hauptprogramm integriert, das die Testlogik und Aufrufstruktur bereitstellt.

Ein unternehmensinterner Compiler überprüft den syntaktischen Aufbau des Codes und übersetzt diesen in eine Dynamic Link Library (DLL), also eine dynamisch geladene Programmbibliothek, die zur Laufzeit vom Testsystem eingebunden wird [THEP23]. Mithilfe eines internen Tools (SW4) kann der erzeugte Code ausgeführt und die Funktionsergebnisse automatisiert überprüft werden. Dabei ruft das Hauptprogramm die generierte Funktion mit allen definierten Eingabeparametern auf und vergleicht die Ausgabewerte mit den erwarteten Resultaten. Stimmen alle Ergebnisse überein, gilt der Testfall als erfolgreich.

Die Ausgaben werden sowohl syntaktisch über den Compiler als auch semantisch über Unit-Tests überprüft, sofern eine gültige Syntax vorliegt. Die zusammenfassende Evaluation erfolgt anhand messbarer Metriken. Diese ermöglichen Rückschlüsse auf die Qualität der LLMs zu ziehen und eine objektive Grundlage für den Vergleich verschiedener Modellanpassungen zu bilden.

# 6 Umsetzung

Dieses Kapitel beschreibt die praktische Umsetzung des in Kapitel 5 vorgestellten Lösungskonzepts. Die Implementierung umfasst die semantische Analyse von TwinCAT-Projekten, die synthetische Generierung von Trainingsdaten, das Fine-Tuning der LLMs sowie das Benchmarking zur Leistungsbewertung. Ergänzend wird in Kapitel 6.5 die Evaluation der Ergebnisse behandelt, bei der sowohl die Qualitätssteigerung der LLMs als auch die Wirtschaftlichkeit des gesamten Prozesses untersucht werden. Zur Interaktion und Steuerung der einzelnen Funktionen wird zudem eine Webanwendung entwickelt, die als zentrale Benutzeroberfläche dient, deren detaillierte Implementierung jedoch aufgrund des Umfangs nicht Hauptbestandteil dieser Arbeit ist.

## 6.1 Semantische Analyse von TwinCAT-Projekten

Die semantische Analyse implementiert den ersten praktischen Schritt für die Lösung. Die Webanwendung stellt Upload-Möglichkeiten bereit, über die einzelne Dateien oder komplette Ordner übertragen werden (vgl. Abbildung A 2). Direkt nach dem Hochladen startet die Analyse im Hintergrund in einem eigenen Thread. Dieses Vorgehen hält die Benutzeroberfläche frei und ermöglicht eine Analyse im Hintergrund, ohne die Sitzung zu blockieren. Für jede Analyse wird eine neue Analysedatei mit einer eindeutigen Bezeichnung erstellt, um sicherzustellen, dass keine Überschreibungen auftreten, wenn TwinCAT-Projekte oder Dateien identische Namen besitzen.

Die Vorverarbeitung liest Quelldokumente, wie beispielsweise TcPOU-Dateien, die als XML-Dateien vorliegen, ein. Die XML-Dateien beinhalten die zwei Abschnitte Declaration und Implementation, die bei der Analyse beachtet werden (vgl. Listing A 1). Im Abschnitt *Declaration* werden alle in der Datei verwendeten Variablen und weiteren Elemente mit ihren Bezeichnern und Datentypen deklariert, während der Abschnitt *Implementation* die logische Umsetzung aus den Quelldokumenten enthält, in der diese Variablen verwendet werden. Aus den gelesenen Inhalten wird eine strukturierte Zwischenrepräsentation erstellt, die die einzelnen Quelldateien den jeweiligen Deklarations- und Implementierungsabschnitten zuordnet. Bei Projektordnern werden alle relevanten Dateien rekursiv durchsucht und die Pfade relativ zur ursprünglichen Ordnerstruktur gespeichert, sodass auch die hierarchische Projektorganisation als Bestandteil der Analyse nachvollziehbar bleibt. Dabei werden ausschließlich für die Analyse relevante Dateien, wie beispielsweise *TcPOU*-, *TcDUT*- oder *TcGVL*-Dateien, berücksichtigt. Nicht benö-

tigte Dateien werden automatisch herausgefiltert und von der weiteren Verarbeitung ausgeschlossen. Fehlerhafte XML-Dateien, die nicht analysierbar sind, werden übersprungen und mit einem Hinweis vermerkt. Die Vorverarbeitung speichert das Ergebnis als JSON-Datei mit einer konsistenten Einrückung, die der Einrückungsstruktur der ursprünglichen Quelldokumente entspricht. Die Datei wird zudem in UTF-8 kodiert, damit Sonderzeichen, Umlaute und internationale Bezeichner korrekt dargestellt werden.

Im nächsten Schritt erfolgt die inhaltliche Auswertung der zuvor erzeugten Zwischenrepräsentation. Dabei werden die zuvor extrahierten Quelltexte anhand definierter Analyseaspekte untersucht, die mithilfe von Prompt-Engineering festgelegt sind. Diese Aspekte definieren, welche syntaktischen, semantischen und strukturellen Merkmale das Modell erkennen und bewerten soll. Dazu gehören beispielsweise charakteristische Sprachkonstrukte der IEC 61131-3, Variablen mit ihren Bezeichnern und Datentypen sowie TwinCAT-spezifische Elemente wie Funktionsbausteininstanzen, Attribute oder Pragmas. Zusätzlich werden Benennungs- und Formatierungskonventionen sowie grundlegende Merkmale des Codes ausgewertet. Das Ausgabeformat des LLMs ist dabei ebenfalls durch Prompt-Engineering festgelegt. Die Ergebnisse werden als JSON-Objekt mit vordefinierten Schlüsseln erzeugt, was eine standardisierte Weiterverarbeitung ermöglicht. Zum Befund speichert die Analyse zusätzliche Metadaten, die den genauen Ursprung im Quellcode kennzeichnen. Dadurch lässt sich im weiteren Verlauf eindeutig nachvollziehen, aus welcher Stelle des Codes ein bestimmter Analyseaspekt abgeleitet wurde.

Falls die Größe eines TwinCAT-Projekts oder einer -Datei das verfügbare Kontextfenster des verwendeten LLMs überschreitet, wird der Analyseprozess automatisch in mehrere Teilanalysen aufgeteilt. Vor Beginn der Analyse erfolgt hierfür eine Tokenisierung der Quelldokumente mithilfe des zum LLM gehörenden Tokenizers. Dabei wird die Anzahl der Tokens ermittelt, um sicherzustellen, dass das Kontextlimit nicht überschritten wird. Falls dieses Limit überschritten wird, segmentiert die Anwendung die Quelldokumente in mehrere logische Abschnitte, die jeweils separat analysiert werden. Jeder Abschnitt erhält eine eigene temporäre Analysedatei mit eindeutiger Kennung, sodass die Ergebnisse nach vollständiger Analyse zusammengeführt werden können. So kann gewährleistet werden, dass umfangreiche Quelldokumente auch innerhalb der technischen Grenzen von LLMs analysiert werden können.

Die Analyse wird durch ein lokales LLM ausgeführt, um den Datenschutz aus TwinCAT-Projekten oder -Dateien zu bewahren. Dafür wird ein lokales LLM aus der Qwen-Familie verwendet. Konkret wird das *Qwen-4B* für das Analyseverfahren eingesetzt [HUGG25e]. Das Modell besitzt ein Kontextfenster von 262.144 Tokens. Dadurch können umfangreiche Analysen von

Quelldokumenten durchgeführt werden. Die Integration des LLM erfolgt über die Transformer-Bibliothek in Python, welche den Zugriff auf vortrainierte LLMs und deren Inferenzfunktionen bereitstellt [HUGG25f]. Das LLM und der zugehörige Tokenizer werden nur bei Bedarf geladen und nach Abschluss der Analyse wieder entladen, wodurch die Nutzung des Grafikspeichers effizient gesteuert wird.

Nach der erfolgreichen Auswertung legt die Anwendung die Ergebnisdatei im Ordner der verarbeiteten Analysedateien ab. Dieser Ordner enthält alle Resultate bereits durchgeführter Analysen von TwinCAT-Projekten und -Dateien. Durch die persistente Speicherung entfällt eine erneute Analyse, wenn ein Projekt oder eine Datei bereits untersucht wurde. In solchen Fällen kann das bestehende Analyseergebnis direkt für die weitere Verarbeitung genutzt werden.

Für jede Analyse wird zusätzlich eine Metadatei erzeugt. Diese beinhaltet Informationen zum Analysefortschritt, zum Typ der Analyse (Projekt oder Datei), zum Zeitpunkt der Verarbeitung sowie zu den Namen der erzeugten Dokumente. Falls eine Analyse fehlschlägt, wird die Fehlermeldung hier hinterlegt. Eine separate Abfrage in der Weboberfläche stellt diese Informationen dar und ermöglicht es, den aktuellen Status jeder Analyse einzusehen, Fehlerquellen nachzuvollziehen und bei Bedarf direkt auf die zugehörigen Ergebnisdateien zuzugreifen.

Nach Abschluss der Analyse wird das Ergebnis in der Weboberfläche visualisiert (vgl. Abbildung A 3). Dort werden die erkannten Analyseaspekte strukturiert dargestellt. Die Benutzeroberfläche bietet zudem die Möglichkeit, einzelne automatisch ermittelte Einträge manuell zu überprüfen, zu korrigieren oder zu ergänzen. Dadurch kann das Analyseergebnis vor dem nächsten Verarbeitungsschritt gezielt angepasst und validiert werden.

## 6.2 Synthetische Generierung von Structured-Text-Trainingsdaten

Aus vorhandenen Analysedateien auf Basis von TwinCAT-Projekten oder -Dateien werden die zentralen Elemente aus der JSON-Datei extrahiert. Auf Basis der extrahierten Informationen zu verschiedenen Aspekten wird ein mehrteiliger Prompt erzeugt. Dieser Prompt beinhaltet die zu beachtenden Vorschriften auf Basis der Analysedateien sowie die Definition des Ausgabeformats und weitere allgemeine Aspekte, die bei der Generierung von ST-Trainingsdaten zu berücksichtigen sind. Für das Ausgabeformat wird erneut ein JSON-Objekt definiert mit den Schlüsseln *Input* und *Output*. Der *Input*-Bereich repräsentiert eine fiktive Nutzeranfrage, die eine Programmieraufgabe oder Problemstellung in natürlicher Sprache beschreibt. Der *Output*-Bereich enthält den zugehörigen synthetisch generierten ST-Code in Bezug auf die fiktive

Nutzeranfrage. Über die Webanwendung können sowohl die zugrunde liegende Analysedatei als auch die gewünschte Anzahl an zu generierenden Trainingsdaten definiert werden.

Um die Varianz und Praxisnähe der Trainingsdaten zu erhöhen, ergänzt die Implementierung zu jedem Prompt gezielte Zufallsanforderungen, etwa kleinere Abwandlungen im Kontrollfluss, bei Datentypen oder im Stil. Dabei werden sowohl allgemeine TwinCAT- und IEC-61131-3-Standardfunktionen als auch projektspezifische Strukturen aus der Analyse automatisch berücksichtigt. Allgemeine ST-Konstrukte wie Schleifen, Timer, Vergleichsoperationen und häufig verwendete Funktionsaufrufe sind bewusst in jeder Prompt-Vorlage enthalten, damit das Modell ein umfassendes Verständnis der Sprache erlernt und universell einsetzbaren Code generieren kann. Ergänzend werden Beckhoff-typische Funktionsbausteine aus relevanten Bibliotheken sowie kundenspezifische Module integriert, sofern diese in der Analysedatei identifiziert wurden. Auf diese Weise entsteht ein ausgewogenes Zusammenspiel aus allgemeinem Sprachwissen und domänenspezifischem Kontext, wodurch das Modell in der Lage ist, sowohl generischen ST-Code korrekt umzusetzen als auch kundenspezifische Anforderungen präzise abzubilden.

Die Generierung von ST-Trainingsdaten ist in zwei verschiedenen Modi implementiert. Im API-Modus wird ein externer LLM-Dienst angesprochen, der auf Basis von verschiedenen Prompt-Ausprägungen synthetische Trainingsdaten erzeugt. Da die Analysedatei ausschließlich abstrahierte Informationen aus TwinCAT-Projekten oder -Dateien enthält, lassen sich daraus keine Rückschlüsse auf interne Kundendaten wie Rohdaten oder projektspezifische Namen ziehen. Dadurch können die Generierungsanfragen gefahrlos an externe LLM-Dienste übermittelt werden. In der Praxis kommt hierfür das Modell *Claude Sonnet 4.5* von Anthropic zum Einsatz, da es sich in Tests des betrachteten Unternehmens als besonders leistungsfähig bei der Generierung syntaktisch korrekten und strukturierten ST-Codes erwiesen hat [ANTH25]. Der API-Modus eignet sich insbesondere für die synthetische Datengenerierung, da dieser Prozess hochparametrisierte LLMs mit einer umfangreichen Wissensbasis benötigt. Diese LLMs sind aufgrund der umfangreichen Parameteranzahl nicht auf handelsüblicher lokaler Hardware ausführbar oder nicht Open-Source zur lokalen Implementierung verfügbar. Im lokalen Modus hingegen wird ein LLM direkt im eigenen System über die Transformer-Bibliothek bereitgestellt. Standardmäßig kommt hier das Modell *Qwen3-30B* zum Einsatz [QWEN25a]. Alternativ kann das spezialisierte Modell *Qwen3-Coder-30B* verwendet werden, das für die strukturierte Codegenerierung angepasst ist [QWEN25b]. Auch bei der synthetischen Trainingsdatengenerierung wird der Tokenizer und das LLM automatisch geladen und nach dem

Generierungsprozess wieder von der Grafikkarte freigegeben. Beide Modi liefern als Antwort reinen Text und können je nach Infrastruktur und Anforderungen flexibel gewählt werden.

Da LLMs gelegentlich Text außerhalb des geforderten JSON-Formats erzeugen, wird die Antwort im Anschluss automatisch bereinigt und in eine gültige Struktur überführt. Dabei wird überprüft, ob ein vollständiges JSON-Objekt in der Antwort vom LLM vorliegt und sich fehlerfrei einlesen lässt. Ist das der Fall, wird das JSON-Objekt übernommen und gespeichert, damit dieses in nachfolgenden Schritten weiterverarbeitet werden kann.

Ein zentrales Qualitätsmerkmal ist die syntaktische Korrektheit des erzeugten ST-Codes. Dafür ist eine Korrekturschleife implementiert, die einen unternehmensinternen Compiler einbindet. Der Compiler wird aus der Python-Anwendung heraus aufgerufen und kompiliert den synthetisch generierten ST-Code, um potenzielle Syntaxfehler zuverlässig zu erkennen. Der Compiler liefert strukturierte Rückmeldungen im JSON-Format mit Angabe der fehlerhaften Zeile, der Fehlerart sowie einer kurzen Beschreibung der möglichen Ursache. Diese Informationen werden anschließend genutzt, um einen spezialisierten Korrektur-Prompt zu erzeugen. Der Prompt enthält den ursprünglichen Code, die vom Compiler gemeldeten Fehler einschließlich Zeilennummer und Fehlertyp sowie eine textuell aufbereitete Empfehlung, wie der Fehler behoben werden kann. Das LLM wird darin explizit angewiesen, den fehlerhaften Code zu korrigieren und ausschließlich ein aktualisiertes JSON-Objekt im identischen Format zurückzugeben. Durch diesen Mechanismus kann das LLM gezielt auf Problemstellen reagieren und die vom Compiler identifizierten Fehler selbstständig beheben. Die Schleife endet, sobald keine weiteren Fehler mehr vom Compiler gemeldet werden oder die maximale Anzahl an Korrekturläufen erreicht ist. Falls die vom Compiler gemeldeten Fehler nach mehreren Korrekturdurchläufen nicht vollständig behoben werden können, wird der ursprüngliche Generierungsprompt leicht angepasst, um ein alternatives Trainingsbeispiel zu erzeugen.

Für die Erzeugung größerer Mengen an Trainingsdaten ist die Generierung für den API-Modus parallelisiert. Eine parallelisierte Task-Steuerung führt mehrere LLM-Aufrufe gleichzeitig aus, wodurch mehrere Prompts parallel verarbeitet werden können. Fehlgeschlagene oder unvollständige Antworten werden verworfen und ein automatischer Wiederholmechanismus ergänzt automatisch neue Aufrufe, bis die gewünschte Anzahl gültiger Datensätze erreicht ist. Um die vom API-Anbieter definierten Nutzungsgrenzen wie die maximalen Ausgabetokens pro Minute einzuhalten, ist die maximale Anzahl paralleler Threads auf zehn begrenzt. Dabei wird insbesondere das Limit von 160.000 Ausgabetokens pro Minute berücksichtigt, um eine Fehlerrückmeldung vom API-Anbieter zu vermeiden [ANTH25b].

Sobald die gewünschte Anzahl vorliegt, werden die einzelnen JSON-Objekte zu einem Datensatz zusammengeführt und als JSON-Array als neuer Datensatz gespeichert. Während des Generierungsprozesses zeigt die Webanwendung den Fortschritt im Generierungsprozess. Der Status des Generierungsprozesses wird beim Start initial angelegt und während des Prozesses aktualisiert, um den Fortschritt bei der Generierung zu protokollieren. Über die Webanwendung erfolgt außerdem die Verwaltung der Datensätze. Generierte Trainingsdaten können heruntergeladen oder gelöscht werden, und bereits vorhandene Datensätze lassen sich manuell hochladen, sofern diese im JSON-Format vorliegen. Während der laufenden Generierung ist es außerdem möglich, die bereits erzeugten Trainingsbeispiele einzusehen. So kann bei qualitativ unzureichenden Ergebnissen der Vorgang frühzeitig abgebrochen werden.

## 6.3 Fine-Tuning von Large Language Models

Der Einstiegspunkt für das Fine-Tuning ist ein Datensatz im JSON-Format, der aus der synthetischen Datengenerierung stammt. Eine Hilfsroutine bereitet daraus zwei JSON-Dateien für Training und Validierung vor, damit das Fine-Tuning-Framework diese als Trainingsdaten verarbeiten kann. Dabei wird zunächst geprüft, ob jedes Element ein *Input*- und *Output*-Feld besitzt. Ungültige und identische Einträge werden übersprungen und nicht für das Fine-Tuning berücksichtigt. Zudem wird beim Fine-Tuning eine Systemnachricht pro Trainingsbeispiel eingebettet, die als konstanter Kontext dient und dem LLM vorgibt, mit welchem Antwortverhalten es auf die im Datensatz enthaltenen Nutzeranfragen reagieren soll. In diesem Fall wird eine standardisierte Systemnachricht ergänzt, die allgemeine Richtlinien für die Generierung von ST-Code enthält, wie beispielsweise Hinweise zur Einhaltung der Syntax oder der Strukturierung. Anschließend wird die Menge im Verhältnis 90/10 in Trainings- und Validierungsanteil geteilt.

Das LLM wird für das Fine-Tuning mit dem Python-Framework *Unsloth* geladen. *Unsloth* ist ein Python-Framework, das speziell für das Fine-Tuning großer LLMs entwickelt wurde und durch optimierte Speicherverwaltung sowie reduzierte Rechenlast einen deutlich schnelleren Fine-Tuning-Prozess ermöglicht, wodurch die Anpassung von LLMs mit erhöhter Parameteranzahl optimiert wird [AWAN24]. In der Standardkonfiguration erfolgt das Fine-Tuning mithilfe des LoRA-Verfahrens und kann direkt so für das Fine-Tuning verwendet werden. Das LLM wird quantisiert in vier Bit geladen, wobei die Modellgewichte von höherer auf geringere Zahlenauflösung umgerechnet werden, um Rechen- und Speicheraufwand zu verringern. Für das Fine-Tuning werden die Projektionen der Zielmodule in den Attention- und Feedforward-Schichten des LLMs definiert, die beim Fine-Tuning angepasst werden sollen. Konkret werden

die Parameterbereiche *q_proj*, *k_proj*, *v_proj*, *o_proj*, *gate_proj*, *up_proj* und *down_proj* angepasst, die nur einen kleinen Teil der Gesamtparameter ausmachen, jedoch maßgeblich das Lernverhalten des LLMs beeinflussen [UNSL25]. Diese Layer steuern die Gewichtung von Attention, Informationsfluss und Aktivierung innerhalb des neuronalen Netzwerks. Die zentralen LoRA-Parameter wie der Rang als Dimension der eingefügten Adapter, Alpha als Verstärkungsfaktor sowie Dropout zur Regularisierung sind konfigurierbar [UNSL25]. Für die verschiedenen *Qwen*-LLM-Varianten werden die empfohlenen LoRA-Parameter in dem Tool automatisch auf Basis des jeweiligen LLMs geladen, entsprechend den Vorgaben der Qwen-Entwickler. Bei Bedarf können diese Parameter jedoch auch manuell über die Webanwendung angepasst werden, um das Fine-Tuning gezielt an spezifische Trainingsanforderungen anzupassen (vgl. Abbildung A 4 - Abbildung A 5).

Die Datensätze der Trainings- und Validierungsdaten werden mit der Hugging Face Dataset-Bibliothek geladen, die ein Standardframework für das effiziente Verwalten und Vorverarbeiten großer Datensammlungen bereitstellt [HUGG25d]. Vor dem Training wird jedes Beispiel in ein einheitliches Textfeld umgewandelt. Dazu wird die Chat-Vorlage des Tokenizers genutzt, welche die einzelnen Kommunikationsrollen System, User und Assistant zu einer linearen Zeichenkette zusammenführt. System steht für die Systemnachricht, User repräsentiert die fiktive Nutzeranfrage und Assistant entspricht der modellgenerierten Antwort in Form des synthetischen ST-Codes. Dadurch entsteht eine homogene Eingabe, ohne dass die interne Rollenlogik des LLMs geändert werden muss.

Das eigentliche Training erfolgt mit dem Supervised Fine-Tuning-Trainer (SFT-Trainer), einem auf das Fine-Tuning spezialisierten Trainingsmodul der Hugging Face-Bibliothek. Wichtige SFT-Trainingsparameter sind ebenso wie die LoRA-Parameter frei konfigurierbar. Die Batchgröße bestimmt die Anzahl der Beispiele pro Schritt, die Gradientenakkumulation fasst mehrere Batches zu einem Optimierungsschritt zusammen, und der Warmup-Anteil erhöht die Lernrate zu Beginn schrittweise, um ein stabiles Anlaufverhalten zu gewährleisten. Die Anzahl der Epochen legt fest, wie oft der Datensatz durchlaufen wird, während Lernrate, Optimierer und Scheduler die Stärke, Methode und zeitliche Anpassung der Gewichtsaktualisierung steuern. Für viele LLMs werden auch diese Parameter bereits für das Fine-Tuning empfohlen und auf die jeweilige Modellarchitektur abgestimmt, können bei Bedarf jedoch über die Webanwendung individuell angepasst werden, um das Training zu optimieren. Standardmäßig wird im bf16-Format trainiert, sofern die Hardware dies unterstützt. Dieses 16-Bit-Format reduziert den Speicherbedarf deutlich, behält jedoch den Wertebereich von 32-Bit-Zahlen bei und sorgt so für ein effizientes und stabiles Training. [HUGG25c]

Die Protokollierung während des Fine-Tuning-Prozesses wird über die Plattform *Weights & Biases* realisiert [H2O25]. Die Plattform ist ein verbreitetes Tool zur Nachverfolgung und Visualisierung von Trainingsmetriken. Die Anbindung kann bei Bedarf vor dem Fine-Tuning aktiviert werden. Während des Fine-Tunings werden regelmäßig Auswertungen auf dem Validierungsdatensatz durchgeführt, und die Logausgaben werden fortlaufend geschrieben, sodass der Fortschritt und die Modellleistung jederzeit nachvollziehbar bleiben.

Nach Abschluss des Fine-Tunings werden die angepassten Gewichte gespeichert. Dafür werden zunächst die angepassten Adapter mit den Basismodell-Gewichten verschmolzen und als zusammengefasste 16-Bit-Gewichte gespeichert. Wenn die Zusammenführung der Gewichte nicht möglich ist, wird das Standardformat verwendet, in dem die vortrainierten Gewichte und die angepassten Adapter separat abgespeichert werden. Die Tokenizer-Dateien werden zusammen mit dem Modell gespeichert, sodass beim späteren Laden automatisch die gleiche Kodierung verwendet wird. Dadurch kann das angepasste LLM direkt für die Inferenz genutzt werden, ohne eine erneute Tokenizer-Konfiguration vorzunehmen.

Die Webanwendung stellt für den Start des Fine-Tuning-Prozesses eine dedizierte Schnittstelle bereit. Beim Start eines Durchlaufs wird ein separater Prozess als Kommandozeilenprogramm gestartet, der die Trainingsumgebung von der Webanwendung trennt. Alle optional eingestellten Parameter aus der Oberfläche der Anwendung werden in Kommandozeilenargumente übersetzt und dem Programm übergeben. Dieses importiert die Trainingsfunktion, setzt die Parameter und startet die Ausführung. Parallel wird eine Logdatei geführt, in der die Konsolenausgabe des Fine-Tuning-Prozesses fortgeschrieben wird. So bleiben auch längere Fine-Tuning-Prozesse nachvollziehbar und stören nicht die weitere Bedienung der Oberfläche.

Die angepassten LLMs werden in der Benutzeroberfläche auf einer Übersichtsseite dargestellt (vgl. Abbildung A 6). Die Webanwendung durchsucht dafür die Ordnerstruktur der angepassten LLMs und extrahiert Metadaten wie Format und Größe. Zusätzlich existiert eine Hilfsfunktion, die aus einem angepassten LLM eine GGUF-Datei erzeugen kann. Dieses Format speichert die Modellgewichte und Metadaten in komprimierter Binärform. Dadurch lassen sich die angepassten LLMs in diversen Anwendungen speichereffizient lokal ausführen. Ergänzend stehen Quantisierungsoptionen zur Verfügung, mit denen die Modellgewichte in geringerer Auflösung gespeichert werden können, um Speicherverbrauch und Rechenlast weiter zu reduzieren, ohne die Modellleistung wesentlich zu beeinträchtigen [CLAR25]. Diese Funktionen werden mithilfe eines Konverters aus dem llama.cpp-Projekt realisiert [KEIT25]. Dieses Projekt

bietet Funktionalitäten zur Umwandlung und Quantisierung von LLMs auf verschiedenen Plattformen. Dadurch entsteht die Möglichkeit, die angepassten LLMs bedarfsgerecht herunterzuladen und flexibel in unterschiedliche Anwendungsbereiche zu integrieren.

## 6.4 Benchmarking von Large Language Models

Die Aufgabenbasis für den Benchmark ist in einer JSON-Datei definiert, wodurch eine strukturierte und leicht erweiterbare Verwaltung der Testfälle ermöglicht wird. Diese Datei enthält 100 diverse Testfälle im Kontext von ST-Anwendungen. Für jede Aufgabe ist der Rückgabetyp, die Eingabeparameter und eine Menge von Testfällen mit Eingaben und erwarteten Ausgaben definiert. Die Testfälle decken ein breites Spektrum typischer TwinCAT-Funktionen und ST-Strukturen ab, darunter elementare Funktionsblöcke wie Timer-, Zähler- und Vergleichsbausteine. Ebenfalls sind Testfälle zu arithmetischen und logischen Operationen sowie Aufrufe häufig verwendeter Bibliotheksfunktionen im Benchmark vorhanden. Dadurch wird sichergestellt, dass der Benchmark sowohl grundlegende ST-Sprachkonstrukte als auch komplexere Beckhoff-spezifische Funktionalitäten beinhaltet. Aus dieser Spezifikation erzeugt die Anwendung zur Laufzeit für jeden Testfall einen präzisen Prompt, der die funktionale Signatur vorgibt und die Aufgabe kompakt beschreibt. Dabei wird das Modell angewiesen, eine vollständige ST-Funktion mit dem Namen *F_Candidate* zu generieren, die exakt den geforderten Rückgabetyp, die Eingabeparameter und die Funktionslogik umsetzt. Auf diese Weise erhalten die LLMs, die getestet werden sollen, eine einheitliche Aufgabenstellung, wodurch die Vergleichbarkeit verschiedener LLMs ermöglicht wird. Außerdem ist durch den Prompt definiert, dass die Funktion durch Marker abgegrenzt wird, um die Funktion zur weiteren Verarbeitung extrahieren zu können.

Die erzeugte Modellantwort wird zunächst bereinigt, damit diese in ein kompilierbares ST-Fragment überführt werden kann. Dafür werden erklärender Text und Formatierungszeichen aus der Modellantwort entfernt, indem der Funktionsbereich, der durch Marker definiert ist, extrahiert wird. Dieser Schritt ist notwendig, da LLMs häufig zusätzlich erklärende Texte in die Antwort einbinden, die beim Verarbeiten durch den Compiler direkt zu Fehlern führen würden.

Aus der Aufgabenbeschreibung und der extrahierten Funktion *F_Candidate* baut die Anwendung ein ausführbares Testprogramm. Dafür wird ein Hauptprogramm *MAIN* aufgebaut, das alle Testfälle sequenziell abarbeitet. Im Deklarationsbereich werden die benötigten Konstanten, Variablen und die Funktion definiert. In der Hauptlogik ruft das Programm die generierte Funktion *F_Candidate* nacheinander mit den Eingaben aller Testfälle auf und vergleicht die

Ergebnisse mit den erwarteten Ausgaben. Weichen Rückgabe und Erwartungswert voneinander ab, wird der Fehlerzähler erhöht und ein entsprechender Statuscode gesetzt. Nach Abschluss aller Testfälle gibt das Programm 0 aus, wenn keine Abweichungen festgestellt wurden, andernfalls 1. Auf diese Weise wird automatisch geprüft, ob die generierte Funktion alle Testfälle korrekt löst.

Die Bewertung erfolgt in einer Kette von Schritten. Für die Bewertung werden zwei unternehmensinterne Tools integriert. Für die syntaktische Qualität wird ein interner Compiler eingesetzt, um sicherzustellen, dass der ST-Code ausgeführt werden kann, während die semantische Qualität durch Unit-Tests überprüft wird. Die Abbildung 6.1 zeigt den automatisierten Prozess zur Feststellung der syntaktischen und semantischen Bewertung einer Aufgabe.

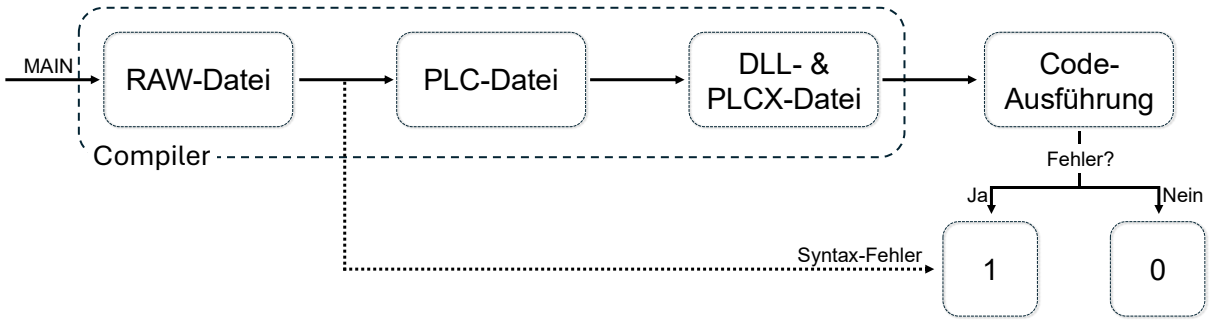


Abbildung 6.1: Prozess zur syntaktischen und semantischen Bewertung von ST-Code

Aus dem Hauptprogramm, das die generierte Funktion beinhaltet, wird zunächst eine Zwischenpräsentation in Form einer RAW-Datei erzeugt. Die RAW-Datei unterscheidet sich dadurch, dass der Deklarationsbereich der Variablen durch das Pragma *__BODY_ST__* und das Programmende durch *__END_ANYPOU* gekennzeichnet werden. Diese Formatierung wird vom Compiler benötigt, um weitere Konvertierungsschritte vorzunehmen. Der unternehmensinterne Compiler liegt als EXE-Datei vor und kann unter Windows direkt über die Konsole gestartet werden, während dieser unter Linux mithilfe der Kompatibilitätsschicht Wine ausgeführt wird, die Windows-Anwendungen in einer angepassten Umgebung lauffähig macht [UBOT25]. Im nächsten Schritt wird über den Aufruf des Compilers eine PLC-Datei erstellt, die den kompilierten ST-Code enthält. Aufgrund verschiedener Ausprägungen der ST-Sprache, die kleinen Abwandlungen der IEC 61131-3 entsprechen, muss die verwendete Sprache beim Aufruf des Compilers mit übergeben werden, da es sonst bei der Kompilierung zu Fehlern kommen kann. Zudem werden die verwendeten TwinCAT-Bibliotheken übergeben, damit der Compiler auf diese zugreifen kann. Sollte der Compiler bereits in diesem Schritt einen Syntaxfehler erkennen, kann der Benchmark nicht weiter ausgeführt werden, da keine ausführbare

Datei erstellt werden kann, und für die Aufgabe wird der Statuscode auf 1 gesetzt. Die Ausgabe der Syntaxfehler erfolgt im JSON-Format, das beim Aufruf ebenfalls als Ausgabeformat der Fehler definiert wird. Dadurch können Syntaxfehler ohne Aufwand weiterverarbeitet werden, um diese in der Webanwendung darzustellen. Wenn die PLC-Datei erfolgreich erstellt werden kann, wird am Ende eine Konfiguration angelegt, die für die spätere Ausführung von essenzieller Bedeutung ist. Die Konfiguration definiert einen zyklischen Ausführungsablauf mit festgelegtem Zeitintervall sowie den Namen des zugehörigen Hauptprogramms, das ausgeführt wird. Das Zeitintervall ist im Kontext der SPS-Architektur relevant, da Programme dort in festen Zyklen ablaufen. Der Name des MAIN-Programms muss dabei exakt angegeben werden, damit die Ausführung korrekt zugeordnet werden kann. Der Compiler erstellt im nächsten Schritt auf Basis der PLC-Datei eine ausführbare DLL- und PLCX-Datei. Die beiden Dateien werden bei der Ausführung des Hauptprogramms, einer weiteren unternehmensinternen EXE-Anwendung, zur Verfügung gestellt. Nach der Abarbeitung liefert die Ausgabe eine Kennziffer als eindeutigen Indikator für das Ergebnis. Dabei steht 0 für eine fehlerfreie Ausführung aller Testfälle, und die Aufgabe kann als syntaktisch und semantisch korrekt angenommen werden, oder 1, was auf eine Abweichung oder einen Fehler im Programmablauf hinweist.

Die Anwendung führt Benchmarks als Hintergrundaufgabe aus, damit die Webanwendung während des Benchmarks bedienbar bleibt. Vor dem Start wird geprüft, ob das ausgewählte LLM für die Inferenz geladen ist, und lädt das LLM bei Bedarf. Danach wird über die Aufgaben iteriert und pro Aufgabe ein eigenes temporäres Arbeitsverzeichnis für die Zwischendateien aus dem Benchmark-Prozess angelegt, die nach dem Durchlauf einer Aufgabe wieder gelöscht werden. Während des Prozesses wird der Status stetig aktualisiert. Zwischenstände und Abschlüsse werden persistent gespeichert, sodass vergangene Läufe in der Webanwendung sichtbar und nachträglich auswertbar bleiben (vgl. Abbildung A 9).

Zusätzlich besteht in der Webanwendung die Möglichkeit, die LLMs direkt über eine Inferenz zu vergleichen (vgl. Abbildung A 7). Dafür können die Basismodelle und die angepassten Modelle über eine Chat-Ansicht geladen werden. Zudem besteht die Möglichkeit, zwei LLMs gleichzeitig zu laden und deren Antworten auf dieselbe Anfrage parallel und übersichtlich nebeneinander zu vergleichen (vgl. Abbildung A 8). Bei der Inferenz lassen sich verschiedene Modellparameter gezielt anpassen, um unterschiedliche Generierungsvarianten miteinander zu vergleichen. Insgesamt kann dadurch die Evaluation gezielt auf spezielle Anfragen und Prompts ausgerichtet werden, um den Erfolg und die Wirkung des Fine-Tuning-Prozesses präzise zu beurteilen.

## 6.5 Evaluation der Ergebnisse

Nach der Implementierung der vier Teilbereiche bildet das Tool eine Möglichkeit zum spezifischen Fine-Tuning von lokalen LLMs. Das Tool analysiert vollständige TwinCAT-Projekte oder -Dateien auf Strukturmerkmale, Richtlinien und Designprinzipien und bündelt diese in einer Analysedatei. Darauf aufbauend generiert das Tool synthetische Trainingsdaten über prompt-gesteuerte Vorlagen, die die Vorgaben aus der Analysedatei berücksichtigen. Zudem wird durch die Integration eines Compilers die syntaktische Korrektheit der Trainingsdaten garantiert. Für das Fine-Tuning integriert das Tool die LoRA-Methode, um lokale LLMs anzupassen. Dabei werden empfohlene modellspezifische Parameter automatisch gesetzt oder bei Bedarf manuell angepasst, um einen optimalen Trainingserfolg zu erzielen. Die Auswertung der Modellleistung erfolgt über einen TwinCAT-spezifischen Benchmark für ST-Code und erfasst die syntaktische und semantische Qualität vor und nach dem Fine-Tuning.

Zur Evaluation der Anpassungsfähigkeit lokaler Large Language Models an TwinCAT-Structured-Text wird ein definierter Testdurchlauf mit dem entwickelten Tool durchgeführt. Dabei wird ein von dem betrachteten Unternehmen bereitgestelltes Beispielprojekt verwendet, das die Beckhoff-Richtlinien vollständig einhält und eine Bandbreite typischer TwinCAT-Funktionen abdeckt. Auf Grundlage der in Kapitel 6.1 beschriebenen semantischen Analyse wird mithilfe eines lokalen LLMs eine Analysedatei erstellt, die die Strukturmerkmale, Richtlinien und Designprinzipien des Beispielprojekts extrahiert. Diese Analysedatei dient als Basis für die in Kapitel 6.2 dargestellte Generierung synthetischer Trainingsdaten, bei der insgesamt 5.000 qualitativ hochwertige Trainingsbeispiele erzeugt werden.

Für die Generierung wird das leistungsstärkste LLM von Anthropic, *Opus 4.1*, eingesetzt, um eine hohe syntaktische und semantische Qualität der erzeugten Daten sicherzustellen. Das LLM verfügt zudem über ein erweitertes Wissen zu TwinCAT-Funktionen und Beckhoff-spezifischen Bibliotheken, wodurch diese in den generierten Trainingsdaten deutlich häufiger und kontextgerechter eingesetzt werden, was die fachliche Relevanz und Varianz der Datensätze zusätzlich erhöht. Die Kosten für die synthetische Generierung von Trainingsdaten werden in Tabelle A 7 dargestellt und mit anderen Modellvarianten von Anthropic verglichen.

Die Generierung mit dem leistungsstärksten LLM, *Opus 4.1*, beläuft sich dabei auf 103 $, wobei sich 60,1 $ bereits auf die Eingabe-Tokens beziehen, die bei jeder Generierung die Definition des Trainingsdatums bestimmen, denn im Durchschnitt besteht eine Anfrage aus 801 Tokens. Die restlichen 42,9 $ beziehen sich auf die Generierung der synthetischen Trainingsdaten, die die fiktiven Nutzeranfragen und den dazugehörigen ST-Code beinhalten. In den

Gesamtwerten sind zudem die Korrekturschleifen des integrierten Compilers enthalten, der fehlerhafte Ausgaben automatisch erkennt und deren Neugenerierung anstößt. Dadurch erhöhen sich sowohl die Input- als auch die Output-Kosten, da bei jeder Korrekturschleife zusätzliche Tokens für die Fehlerbeschreibung, Rückmeldung und erneute Codeausgabe verarbeitet werden. Dieser Anstieg des Tokenverbrauchs zeigt sich auch bei leistungsschwächeren Modellen, da diese erfahrungsgemäß häufiger syntaktische Fehler erzeugen und somit mehr Iterationen zur Korrektur erforderlich sind. Zum Einordnen der Größenordnung liegen die Kosten bei den weiteren LLMs von Anthropic deutlich niedriger. Für *Sonnet 4.5* betragen diese insgesamt 23 $, für das leistungsschwächste Modell *Haiku 4.5* etwa 7,50 $. Diese Modelle sind zwar kostengünstiger in der Datengenerierung, weisen jedoch geringere Kenntnisse über TwinCAT-spezifische Strukturen, Systemfunktionen und Bibliotheken auf. Infolgedessen erzeugen sie weniger komplexe Programmstrukturen und nutzen vorhandene Funktionsbausteine nicht in gleicher Effizienz und Tiefe wie *Opus 4.1*, was die syntaktische und semantische Qualität der Trainingsdaten einschränkt.

Die gesamte Generierung der 5.000 Trainingsdaten dauert bei paralleler Ausführung mit zehn Threads rund 45 Minuten, was einer durchschnittlichen Erzeugungsrate von etwa 111 synthetischen Trainingsbeispielen pro Minute entspricht. Dies unterstreicht die hohe Effizienz des Generierungsprozesses trotz der Verwendung eines komplexen Modells und umfangreicher Validierungsschleifen.

Das Fine-Tuning wird auf lokale LLMs mit einer diversen Anzahl an Parametern angewendet, um den Einfluss des Fine-Tunings bei variierender Parameteranzahl zu messen. Evaluierte LLMs sind *Qwen-4B*, *Qwen-14B* und das auf Programmieranfragen spezialisierte *Qwen-30B-Coder*-LLM. Diese Modellreihen zeigen sich im Rahmen interner Evaluierung als besonders leistungsfähig im Umgang mit TwinCAT-ST-Code und bieten daher ein optimales Potenzial, durch gezieltes Fine-Tuning zu leistungsstarken ST-LLMs weiterentwickelt zu werden. Die Trainingskonfiguration folgt den in Kapitel 6.3 beschriebenen Prinzipien. Für das Fine-Tuning werden die dort dargelegte Auswahl an Schichten trainiert und eine Lernrate von 2e-4 verwendet. Der Dropout-Wert wird durch die Unsloth-Bibliothek automatisch berücksichtigt, sodass eine manuelle Parametrisierung entfällt. Die Adapterkapazitäten werden durch Kombinationen der Parameter $r$ (Rang) und $\alpha$ (Skalierungsfaktor) in den Konfigurationen *(8/16)*, *(16/32)*, *(32/64)* und *(64/128)* variiert, um den Einfluss unterschiedlicher LoRA-Konfigurationen auf die Modellleistung systematisch zu untersuchen. Höhere Werte führen zu einem deutlich erhöhten Speicherbedarf, wodurch die Experimente durch die verfügbaren 96 GB VRAM der eingesetzten NVIDIA RTX 6000 Pro begrenzt sind. Exemplarisch ergeben sich bei *Qwen-4B* für die

Kombination *r = 64* und *α = 128* etwa 132 Millionen trainierbare Parameter, wohingegen die gleiche Variante beim Qwen-30B-Coder-LLM bereits über drei Milliarden Parameter beinhaltet (vgl. Tabelle A 2 - Tabelle A 4). Empirisch zeigt sich ein Optimum bei zwei Trainingsepochen. Der Übergang von einer auf zwei Epochen steigert Syntax- und Semantikqualität messbar, jedoch bringt eine dritte Epoche keinen weiteren Zugewinn an Qualität und erhöht das Risiko einer Überanpassung an synthetische Muster. Die Batchgröße beträgt zwei, die Gradientenakkumulation erfolgt über vier Schritte, und die Warmup-Phase umfasst fünf Schritte. Als Optimierer wird AdamW in 8-Bit-Präzision verwendet, ergänzt durch einen linearen Lernraten-Scheduler. Diese Parameter sind speziell auf das Fine-Tuning der *Qwen*-Modelle abgestimmt und werden automatisch gesetzt, um eine stabile Anpassung sicherzustellen.

Das Fine-Tuning wird mit den definierten Parametern und den ausgewählten LLMs durchgeführt und durch den in Kapitel 6.4 beschriebenen Benchmark auf syntaktische und semantische Qualität geprüft. Die Tabelle 6.1 zeigt die bedeutendsten Ergebnisse des durchgeführten Vergleichs und belegt zugleich die Wirksamkeit des Fine-Tuning-Prozesses auf die lokalen LLMs.

| **Modellfamilie** | **Variante** | **Trainierte Parameter** | **Syntaktische Qualität** | **Semantische Qualität** |
|---|---|---|---|---|
| **Qwen-4B** | Basismodell | / | 54 | 49 |
| | r=64, α=128 | 132 Millionen | **67** | **54** |
| **Qwen-14B** | Basismodell | / | 58 | 47 |
| | r=64, α=128 | 256 Millionen | **71** | **59** |
| **Qwen-30B-Coder** | Basismodell | / | 68 | 59 |
| | r=16, α=32 | 843 Millionen | **73** | **66** |

Tabelle 6.1: Vergleich der Fine-Tuning-Ergebnisse verschiedener Modellfamilien

Bei *Qwen-4B* verbessert die Variante *(64/128)* die syntaktische Qualität von 54 Punkten auf 67 Punkte (+ 24,1 %) und die semantische Qualität von 49 Punkten auf 54 Punkte (+ 10,2 %) bei gleichbleibender Generierungsleistung von 82,7 Tokens pro Sekunde (vgl. Tabelle A 2). Das *Qwen-14B*-Basismodell zeigt in der Basisvariante eine leicht bessere syntaktische Qualität als das parameterschwächere *Qwen-4B*-LLM, weist jedoch bei der semantischen Prüfung mehr Fehler auf als das *Qwen-4B*-LLM. Durch das Fine-Tuning mit der Variante *(64/128)* kann

die semantische Qualität des *Qwen-14B*-LLMs von 47 Punkten auf 59 Punkte (+ 25,5 %) erheblich verbessert werden (vgl. Tabelle A 3). Auch die syntaktische Qualität kann bei diesem LLM von 58 Punkten auf 71 Punkte (+ 22,4 %) optimiert werden. Die Anzahl an Tokens, die generiert wird, liegt bei dieser Modellvariante bei 40,0 Tokens pro Sekunde. Das parameterstärkste *Qwen-30B-Coder*-LLM zeigt in der Basisvariante die beste Qualität und kann nach dem Fine-Tuning im Vergleich zu den Fine-Tuning-Ergebnissen der anderen angepassten LLMs eine geringere Verbesserung der Qualität erreichen. Das *Qwen-30B-Coder*-LLM erzielt die höchste Qualität durch das Fine-Tuning der Variante *(16/32)*, wodurch 843 Millionen Parameter angepasst werden. Die syntaktische Qualität kann von 68 Punkten auf 73 Punkte (+ 7,4 %) und die semantische Qualität von 59 Punkten auf 66 Punkte (+ 11,9 %) gesteigert werden, bei einer Generierungsleistung von 18,7 Tokens pro Sekunde (vgl. Tabelle A 4). Die benötigte Zeit für das Fine-Tuning dieser Varianten ist in Tabelle A 5 dargestellt. Insgesamt zeigen die Ergebnisse, dass insbesondere das Fine-Tuning bei LLMs mit einer geringen und mittleren Anzahl an Parametern die stärksten Verbesserungen in Bezug auf syntaktische und semantische Qualität hervorbringt. Allgemein zeigt das *Qwen-30B-Coder*-LLM die beste absolute Qualität unter den angepassten lokalen LLMs.

Der externe Vergleich mit eingesetzten Cloud-Modellen kontextualisiert die Ergebnisse (vgl. Tabelle A 6). Das verbreitet eingesetzte *Sonnet 4.5* LLM erreicht eine syntaktische Qualität von 83 Punkten und eine semantische Qualität von 71 Punkten, wohingegen das stärkste LLM aus der Anthropic-Modellfamilie *Opus 4.1* eine syntaktische Qualität von 93 Punkten und eine semantische Qualität von 80 Punkten erreicht. Die besten lokalen Ergebnisse nach dem Fine-Tuning bleiben zwar unterhalb von *Sonnet 4.5*, reduzieren jedoch den Abstand insbesondere im semantischen Bereich deutlich.

Auf Basis dieser Evaluation zeigt sich, dass Cloud-LLMs dieser Größenordnung derzeit noch eine höhere absolute Leistungsfähigkeit aufweisen und lokale LLMs durch alleiniges Fine-Tuning nicht auf die gleiche Qualität gebracht werden können. Dennoch stellt das Fine-Tuning eine wirkungsvolle Methode dar, lokale LLMs kundenspezifisch zu adaptieren, die Qualität gezielt zu steigern und gleichzeitig unabhängig, datenschutzkonform und domänenspezifisch einsetzbar zu machen (vgl. Abbildung A 10).

# 7 Zusammenfassung und Ausblick

Die Arbeit entwickelt und validiert ein Tool, das lokale LLMs für TwinCAT-ST durch Fine-Tuning automatisiert anpasst. Das System vereint mehrere funktional aufeinander abgestimmte Komponenten, indem es zunächst bestehende TwinCAT-Projekte oder -Dateien semantisch analysiert, um deren Strukturmerkmale, Richtlinien und Designprinzipien zu erfassen. Auf dieser Grundlage werden synthetische ST-Trainingsdaten generiert und mithilfe integrierter Validierungsmechanismen hinsichtlich der Syntax überprüft. Anschließend erfolgt das Fine-Tuning lokaler LLMs mittels des LoRA-Verfahrens, wodurch sich die Modellparameter effizient an die Domäne der SPS-Programmierung anpassen lassen. Abschließend bewertet ein Benchmarking-Prozess die Leistungsfähigkeit der LLMs anhand syntaktischer und semantischer Prüfungen. Die definierten Anforderungen an das Tool **A1–A8** sind durch die Umsetzung vollständig erfüllt.

Die Ergebnisse belegen, dass domänenspezifisches Fine-Tuning die Qualität bei der Generierung von ST-Code signifikant verbessert. Durch Fine-Tuning kann die syntaktische Qualität von LLMs in Bezug auf ST-Code um durchschnittlich 18,0 % gesteigert werden. Ähnliche Zuwächse zeigen sich bei der Steigerung der semantischen Qualität, die im Durchschnitt um 15,9 % verbessert wird. Die Bewertung mittels Compiler- und Unit-Tests bestätigt eine messbare Qualitätssteigerung, während ergänzende Kosten- und Generierungsanalysen den effizienten Ressourceneinsatz der Lösung aufzeigen.

Für das betrachtete Unternehmen bedeuten die Ergebnisse, dass lokales Fine-Tuning mit synthetischen Trainingsdaten den Bedarf an realen Kundendaten reduziert, wodurch Vertraulichkeit gewahrt bleibt. Zudem wird dadurch die Passgenauigkeit für das kundenspezifische Anpassen lokaler LLMs ermöglicht, sodass die Generierung von ST-Code den Kundenanforderungen gerecht wird.

Eine Weiterentwicklung des Tools könnte darin bestehen, den Generierungsprozess um zusätzliche Wissensquellen zu erweitern. Über Retrieval-Augmented Generation (RAG) ließen sich TwinCAT-Dokumentationen, Bibliotheksbeschreibungen oder interne Codebeispiele direkt in die Prompterstellung einbinden, um die Kontexttreue zu erhöhen und die Qualität der Trainingsdaten weiter zu verbessern. Parallel dazu ließe sich der Benchmarkbereich gezielt ausbauen, um für das betrachtete Unternehmen einen umfangreichen und einheitlichen Bewertungsstandard zu etablieren. Durch die Erweiterung der bestehenden Benchmark-Testfälle könnten verschiedene LLMs künftig systematisch und vergleichbar auf syntaktische und semantische Qualität geprüft werden, um leistungsfähige LLMs zu identifizieren.

# 8 Literaturverzeichnis


**[AHMA25]** **Ahmad et al., Wasi Uddin**: *OpenCodeInstruct: A Large-scale Instruction Tuning Dataset for Code LLMs*; NVIDIA (Hrsg.), 2025, Santa Clara, CA 15213, USA
https://doi.org/10.48550/arXiv.2504.04030, Zugriff am 30. September 2025

**[ALAC20]** **Alarcon, Nefi**: Nvidia Developer, Juli 2020 [Online]
https://developer.nvidia.com/blog/openai-presents-gpt-3-a-175-billion-parameters-language-model/, Zugriff am 23. September 2025

**[ANTH25]** **Anthropic,** : Anthropic - Introducing Claude Sonnet 4.5, September 2025 [Online]
https://www.anthropic.com/news/claude-sonnet-4-5, Zugriff am 15. Oktober 2025

**[ANTH25b]** **Anthropic,** : Claude Docs - Rate limits, 2025 [Online]
https://docs.claude.com/en/api/rate-limits, Zugriff am 15. Oktober 2025

**[AWAN24]** **Awan, Abid Ali**: Datacamp, Oktober 2024 [Online]
https://www.datacamp.com/tutorial/unsloth-guide-optimize-and-speed-up-llm-fine-tuning, Zugriff am 15. Oktober 2025

**[BECK25a]** **Beckhoff Automation GmbH & Co. KG,** : Beckhoff Information System [Online]
https://infosys.beckhoff.com/index.php?content=../content/1031/te1010_tc3_realtime_monitor/6828869003.html&id=, Zugriff am 22. September 2025

**[BECK25b]** **Beckhoff Automation GmbH & Co. KG,** : Beckhoff Information System [Online]
https://infosys.beckhoff.com/index.php?content=../content/1031/tc3_io_intro/1842809099.html&id=, Zugriff am 22. September 2025

**[BECK25c]** **Beckhoff Automation GmbH & Co. KG,** : Beckhoff Information System [Online]
https://infosys.beckhoff.com/index.php?content=../content/1031/tcplccontrol/925248523.html&id=, Zugriff am 22. September 2025

**[BECK25d]** **Beckhoff Automation GmbH & Co. KG,** : Beckhoff Information System [Online]
https://infosys.beckhoff.com/index.php?content=../content/1031/tcquickstart/5281654411.html&id=, Zugriff am 22. September 2025

**[BECK25e]** **Beckhoff Automation GmbH & Co. KG,** : Beckhoff Information System [Online]
https://infosys.beckhoff.com/index.php?content=../content/1031/tc3_plc_intro/2530284939.html&id=, Zugriff am 22. Semptember 2025

**[BECK25f]** **Beckhoff Automation GmbH & Co. KG,** : Beckhoff Information System [Online]

https://infosys.beckhoff.com/index.php?content=../content/1031/tc3_plc_intro/2530279563.html&id=, Zugriff am 22. September 2025

**[BECK25g]** **Beckhoff Automation GmbH & Co. KG,** : Beckhoff Information System [Online] https://infosys.beckhoff.com/index.php?content=../content/1031/tc3_plc_intro/2530274187.html&id=, Zugriff am 22. September 2025

**[BECK25h]** **Beckhoff Automation GmbH & Co. KG,** : Beckhoff Information System [Online] https://infosys.beckhoff.com/index.php?content=../content/1031/tcplclib_tc2_standard/73975051.html&id=, Zugriff am 22. September 2025

**[BECK25i]** **Beckhoff Automation GmbH & Co. KG,** : Beckhoff Information System [Online] https://infosys.beckhoff.com/english.php?content=../content/1033/tcplclib_tc2_mc2/index.html&id=, Zugriff am 29. September 2025

**[BECK25j]** **Beckhoff Automation GmbH & Co. KG,** : Beckhoff - Vom Chatbot zum intelligenten Agenten: TwinCAT Chat wird zu TwinCAT CoAgent, 2025 [Online] https://www.beckhoff.com/de-de/produkte/automation/twincat-projekte-mit-ki-unterstuetztem-engineering/, Zugriff am 01. Oktober 2025

**[BELC25]** **Belvic, Ivan und Stryker, Cole**: IBM - RAG vs. fine-tuning vs. prompt engineering, 2025 [Online] https://www.ibm.com/think/topics/rag-vs-fine-tuning-vs-prompt-engineering, Zugriff am 24. September 2025

**[CLAR25]** **Clark, Bryan**: IBM - Was ist Quantisierung?, 2025 [Online] https://www.ibm.com/de-de/think/topics/quantization, Zugriff am 17. Oktober 2025

**[CONT25]** **Contact and Coil,** : Contact and Coil [Online] https://www.contactandcoil.com/twincat-3-tutorial/writing-your-own-functions-and-function-blocks/, Zugriff am 22. September 2025

**[FECH25]** **fecher GmbH,** : fecher [Online] https://www.modernizing-applications.de/it-glossar/programmbibliothek-definition/, Zugriff am 22. September 2025

**[GADE25]** **Gadesha, Vrunda; Kavlakoglu, Eda; Winland, Vanna**: IBM - What is chain of thought (CoT) prompting?, 2025 [Online] https://www.ibm.com/think/topics/chain-of-thoughts#1774455706, Zugriff am 2025. September 24

**[GOOG25]** **Google Cloud,** : Cloud Google - Prompt Engineering: Übersicht und Leitfaden, 2025 [Online] https://cloud.google.com/discover/what-is-prompt-engineering?hl=de, Zugriff am 24. September 2025

**[GRAM25]** **Gramsch, Maria**: Basic Thinking, September 2025 [Online] https://www.basicthinking.de/blog/2025/09/05/synthetische-daten-ki/, Zugriff am 25. September 2025

**[H2O25]** **H2O,** : H2O.ai - What are Weights and Biases?, 2025 [Online] https://h2o.ai/wiki/weights-and-biases/, Zugriff am 15. Oktober 2025

**[HASE25]** **Hasenäcker, Jana Verena**: Thesius [Online] https://thesius.de/thema-abschlussarbeit/2553413, Zugriff am 23. September 2025

**[HENNE25]** **Henneken, Stefan**: IEC 61131-3: Unit-Tests [Online] https://stefanhenneken.net/2017/11/14/iec-61131-3-unit-test/, Zugriff am 22. September 2025

**[HU21]** **Hu, Edward J. et al.**: *LoRA: Low-Rank Adaptation of Large Language Models*; 2021 https://doi.org/10.48550/arXiv.2106.09685, Zugriff am 25. Semptember 2025

**[HUGG25a]** **HuggingFace,** : Hugging Face - Safetensors, 2025 [Online] https://huggingface.co/docs/safetensors/index, Zugriff am 09. Oktober 2025

**[HUGG25b]** **HuggingFace,** : Hugging Face - GGUF, 2025 [Online] https://huggingface.co/docs/hub/gguf, Zugriff am 09. Oktober 2025

**[HUGG25c]** **Hugging Face,** : Hugging Face - SFT Trainer, 2025 [Online] https://huggingface.co/docs/trl/sft_trainer, Zugriff am 15. Oktober 2025

**[HUGG25d]** **HuggingFace,** : Hugging Face - Datasets, 2025 [Online] https://huggingface.co/docs/datasets/index, Zugriff am 17. Oktober 2025

**[HUGG25e]** **Hugging Face,** : Hugging Face - Qwen3-4B-Thinking-2507, 2025 [Online] https://huggingface.co/Qwen/Qwen3-4B-Thinking-2507, Zugriff am 2. November 2025

**[HUGG25f]** **Hugging Face,** : Hugging Face - Transformers, 2025 [Online] https://huggingface.co/docs/transformers/de/index, Zugriff am 02. November 2025

**[IEC25]** **IEC,** : IEC 611313 - Programmable controllers – Part 3: Programming languages, 40. Auflage, VDE VERLAG GmbH, 2025-05, Geneva, Switzerland, ISBN: 978-2-8327-0436-3 https://www.vde-verlag.de/iec-normen/255113/iec-61131-3-2025.html, Zugriff am 22. September 2025

**[JIME25]** **Jimenez et al., Calos E.**: 2024 https://doi.org/10.48550/arXiv.2310.06770, Zugriff am 29. September 2025

**[JURA25a]** **Jurafsky, Dan und Martin, James H.**: *Speech and Language Processing (3rd ed. draft) - Large Language Models*; Stanford University (Hrsg.), 2025, California [Online] https://web.stanford.edu/~jurafsky/slp3/7.pdf, Zugriff am 23. September 2025

**[JURA25b]** **Jurafsky, Dan und Martin, James H.**: *Speech and Language Processing (3rd ed. draft) - Words and Tokens*; Stanford University (Hrsg.), 2025, California [Online] https://web.stanford.edu/~jurafsky/slp3/2.pdf, Zugriff am 23. September 2025

**[JURA25c]** **Jurafsky, Dan und Martin, James H.**: *Speech and Language Processing (3rd ed. draft) - Embeddings*; Stanford University (Hrsg.), 2025, California [Online] https://web.stanford.edu/~jurafsky/slp3/5.pdf, Zugriff am 23. September 2025

**[JURA25d]** **Jurafsky, Dan und Martin, James H.**: *Speech and Language Processing (3rd ed. draft) - Neural Networks*; Stanford University (Hrsg.), 2025, California [Online] https://web.stanford.edu/~jurafsky/slp3/6.pdf, Zugriff am 23. September 2025

**[JURA25e]** **Jurafsky, Dan und Martin, James H.**: *Speech and Language Processing (3rd ed. draft) - Transfomers*; Stanford University (Hrsg.), 2025, California [Online] https://web.stanford.edu/~jurafsky/slp3/8.pdf, Zugriff am 23. September 2025

**[KEIT25]** **Keita, Zoumana**: Datacamp - Llama.cpp Tutorial, Januar 2025 [Online] https://www.datacamp.com/de/tutorial/llama-cpp-tutorial, Zugriff am 17. Oktober 2025

**[KRÜG25]** **Krüger, Maria**: Linvelo - Synthetic Data for Industrial AI, August 2025 [Online] https://linvelo.com/synthetic-data-for-industrial-ai/, Zugriff am 25. September 2025

**[LI24]** **Li et al., Junjie**: *An Exploratory Study on Fine-Tuning Large Language Models for Secure Code Generation*; Concordia University; Queen's University; Delhi Technological University (Hrsg.), 2024 https://doi.org/10.48550/arXiv.2408.09078, Zugriff am 30. September 2025

**[MEYE21]** **Meyer, Uwe**: Compilerbau - Grundkurs, 2. Auflage, Rheinwerk Computing, 2024, ISBN: 978-3-8362-9671-7 https://www.rheinwerk-verlag.de/grundkurs-compilerbau/?srsltid=AfmBOooi1qHgeeAHsMVGB5cXYKDZ20nbxQfqH1gdK4OSH9r6OhyVpoUo, Zugriff am 25. September 2025

**[MORR23]** **Morrison, Ryan**: Tech Monitor - The majority of AI training data will be synthetic by next year, says Gartner, August 2023 [Online] https://www.techmonitor.ai/digital-economy/ai-and-automation/ai-synthetic-data-edge-computing-gartner?cf-view, Zugriff am 25. September 2025

**[MURE24]** **Murel, Jacob und Kavlakoglu, Eda**: IBM - Was ist Datenerweiterung?, Mai 2024 [Online] https://www.ibm.com/de-de/think/topics/data-augmentation, Zugriff am 25. September 2025

**[NADA25]** **Nadas, Mihai und Diosan, Laura**: *SYNTHETIC DATA GENERATION USING LARGE LANGUAGE*; Faculty of Mathematics and Computer Science, Babe¸s-Bolyai University (Hrsg.), 2025, Cluj-Napoca, Romania https://doi.org/10.48550/arXiv.2503.14023, Zugriff am 30. September 2025

**[NIKO21]** **Nikolenko, Sergey I.**: Synthetic Data for Deep Learning, Springer Verlag, 2021, San Francisco, CA, USA, ISBN: 978-3-030-75178-4 https://link.springer.com/book/10.1007/978-3-030-75178-4, Zugriff am 25. September 2025

**[PASI22]** **Pasieka, Manuel**: Mostly AI - A comparison of synthetic data generation methods, September 2022 [Online] https://mostly.ai/blog/comparison-of-synthetic-data-generation-methods, Zugriff am 25. September 2025

**[QWEN25a]** **Qwen,** : GitHub - , Oktober 2025 [Online] https://github.com/QwenLM/Qwen3, Zugriff am 15. Oktober 2025

**[QWEN25b]** **Qwen,** : GitHub - Qwen3-Coder, Juli 2025 [Online] https://github.com/QwenLM/Qwen3-Coder, Zugriff am 15. Oktober 2025

**[RAUF24]** **Rauf, Zohaib**: A beginners guide to fine tuning LLM using LoRA, Februar 2024 [Online] https://zohaib.me/a-beginners-guide-to-fine-tuning-llm-using-lora/, Zugriff am 30. September 2025

**[RAVI25]** **Ravi et al., Ravin**: *LLMLOOP: Improving LLM-Generated Code and Tests through Automated Iterative Feedback Loops*; University of Auckland (Hrsg.), 2025, Auckland, New Zealand , DOI: 10.1109/ICSME64153.2025.00109 https://ieeexplore.ieee.org/document/11185878

**[SAND25]** **Sandgarden,** : Sandgarden - What is LoRA? A Guide to Guide Fine-Tuning LLMs Efficiently with Low-Rank Adaptation, 2025 [Online] https://www.sandgarden.com/learn/lora-low-rank-adaptation, Zugriff am 30. September 2025

**[SCHI25]** **Schindler, Christian und Rausch, Andreas**: *LLM-Based Design Pattern Detection*; Institute for Software and Systems Engineering (Hrsg.), 2025, Clausthal-Zellerfeld, Germany https://doi.org/10.48550/arXiv.2502.18458, Zugriff am 30. September 2025

**[THEP23]** **thePower,** : thePower - Was ist eine Dynamic Link Library?, Juli 2023 [Online] https://www.thepowermba.com/de/blog/was-ist-eine-dynamic-link-library, Zugriff am 10. Oktober 2025

**[UBOT25]** **Ubot,** : Ubuntuusers - Wine, April 2025 [Online] https://wiki.ubuntuusers.de/Wine/, Zugriff am 17. Oktober 2025

**[UNSL25]** **Unsloth,** : Unsloth - LoRA Hyperparameters Guide, August 2025 [Online] https://docs.unsloth.ai/get-started/fine-tuning-llms-guide/lora-hyperparameters-guide, Zugriff am 17. Oktober 2025

**[VASW17]** **Vaswani, Ashish et al.**: *Attention Is All You Need*; 2017, Long Beach, CA, USA https://doi.org/10.48550/arXiv.1706.03762, Zugriff am 23. Semptember 2025

**[WANG25]** **Wang, Dannong**: *FinLoRA: Benchmarking LoRA Methods for Fine-Tuning LLMs on Financial Datasets*; Rensselaer Polytechnic Institute; Columbia

University; Stevens Institute of Technology (Hrsg.), 2025, Columbia https://doi.org/10.48550/arXiv.2505.19819, Zugriff am 30. September 2025

**[WEIN25]** **Wein, Marcel**: WeinSolutions [Online] https://weinsolutions.de/das-eva-prinzip-in-der-sps-steuerung-einfach-erklaert/, Zugriff am 22. September 2025

**[XION20]** **Xiong et al., Ruibin**: *On Layer Normalization in the Transformer Architecture*; 2020 https://doi.org/10.48550/arXiv.2002.00474, Zugriff am 30. September 2025

**[ZARE25]** **Zarecki, Iris**: K2View Blog - Prompt engineering vs fine-tuning: Understanding the pros and cons, 2025 [Online] https://www.k2view.com/blog/prompt-engineering-vs-fine-tuning/#Prompt-engineering-defined, Zugriff am 24. September 2025

**[ZHEN25]** **Zhenyu et al., Pan**: *Do Code LLMs Understand Design Patterns?*; 2025 https://doi.org/10.48550/arXiv.2501.04835, Zugriff am 30. September 2025

**[ZIYA25]** **Ziyang et al., Luo**: *WizardCoder: EMPOWERING CODE LARGE LANGUAGE MODELS WITH EVOL-INSTRUCT*; Microsoft (Hrsg.), 2025, Hong Kong Baptist University https://doi.org/10.48550/arXiv.2306.08568, Zugriff am 30. September 2025

**[ZÜHL11]** **Zühlke, Torsten**: *Compiler*; Universität Hamburg (Hrsg.), 2011, Hamburg Ausarbeitung zum Proseminar [Online] https://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2011/cgk11-zuehlke-compiler-ausarbeitung.pdf, Zugriff am 25. September 2025

# 9 Softwareverzeichnis

## Open-Source-Software | Software-Projekte

| Referenz im Text | Beschreibung des Software-Projekts |
|---|---|
| SP1 | Name: xxx<br>Bezugsquelle / URL: xxx<br>Lizenzname: xxx<br>Version oder Zugriffsdatum: xxx<br>Begründung/Verwendungsform: xxx |

## Verwendete Software-Werkzeuge

| Referenz im Text | Beschreibung des Software-Werkzeugs |
|---|---|
| SW1 | Name: TwinCAT 3.1 – eXtended Automation Engineering (XAE)<br>Version: Version 3.1, Build 4026.19<br>Verwendungszweck: Erstellung und Ausführung von Test- und Demo-Projekten zur Validierung des entwickelten Tools |
| SW2 | Name: Visual Studio Code<br>Version: 1.105.1<br>Verwendungszweck: Implementierung der entwickelten Softwarekomponenten des Tools |
| SW3 | Name: Solar-Compiler (unternehmensintern)<br>Version: nicht versioniert<br>Verwendungszweck: Kompilierung von ST-Code zur syntaktischen Überprüfung und Vorverarbeitung im Rahmen der Codeausführung |
| SW4 | Name: Dert-Codeausführung (unternehmensintern)<br>Version: nicht versioniert<br>Verwendungszweck: Ausführung von ST-Programmcode |
| SW5 | Name: ChatGPT<br>Version: GPT-5-2025-08-07<br>Verwendungszweck: Unterstützung bei der sprachlichen Überarbeitung von Texten, einschließlich Korrektur von Rechtschreibung, Zeichensetzung, Satzbau und Optimierung von Formulierungen |

# 10 Anhang

Der Anhang ergänzt die in den Kapiteln dargestellten Inhalte durch technische Erläuterungen, Details und zusätzliche Abbildungen. Kapitel 10.1 beschreibt die zentralen Fine-Tuning-Parameter des LoRA-Verfahrens, Kapitel 10.2 erläutert die zugrunde liegende Transformer-Architektur, und Kapitel 10.3 zeigt exemplarisch die XML-Struktur einer TwinCAT-POU-Datei. Darauf aufbauend veranschaulichen die Kapitel 10.4 bis 10.6 die praktische Umsetzung des entwickelten Tools anhand der Weboberfläche, der erzielten Modellergebnisse sowie eines exemplarischen Generierungsvergleichs zwischen Basismodell und Fine-Tuned-Modell.

## 10.1 Fine-Tuning-Parameter für das LoRA-Verfahren

Für effektives Fine-Tuning von LLMs sind einige Parameter vor dem Fine-Tuning zu definieren. Die Tabelle A 1 zeigt die zentralen Trainingsparameter, die beim PEFT-/LoRA-Verfahren festgelegt werden, um das domänenspezifische Fine-Tuning optimal zu gestalten.

| **Parameter** | **Beschreibung** |
|---|---|
| **Target_modules (Zielmodule)** | Definiert die Modellschichten, in denen die LoRA-Adapter integriert werden. Diese Adapter werden in die linearen Projektionsschichten der Self-Attention (*q_proj, k_proj, v_proj, o_proj*) sowie in die Feedforward-Komponenten (*gate_proj, up_proj, down_proj*) eingefügt. Durch die Anpassung dieser Module wird die Gewichtung von Attention, Informationsfluss und Aktivierung innerhalb des neuronalen Netzwerks beeinflusst. |
| **Rang (r)** | Gibt die Dimension der Low-Rank-Matrizen an und bestimmt, wie stark die zusätzlichen Adapter in die Modelltransformation eingreifen. Ein höherer Rang erhöht die Anpassungsfähigkeit, aber auch den Rechenaufwand. |
| **LoRA_Alpha (α)** | Skalierungsfaktor, der die Gewichtung der Low-Rank-Adapter im Verhältnis zu den eingefrorenen Basismodellparametern steuert. Größere Werte führen zu stärkeren Korrekturen, können aber auch zu Instabilität führen. |

| | |
|---|---|
| **Epochenanzahl** | Gibt an, wie oft das Trainingsdataset vollständig durchlaufen wird. Eine höhere Epochenzahl kann die Modellanpassung verbessern, erhöht aber das Risiko des Übertrainings. |
| **Lernrate** | Bestimmt die Schrittweite, mit der die Gewichte während der Optimierung angepasst werden. Eine zu hohe Lernrate kann das Training instabil machen, eine zu niedrige verlangsamt den Lernfortschritt. |
| **Dropout** | Reguläre Mechanismus, der zufällig Anteile der neuronalen Aktivierungen während des Trainings deaktiviert, um Überanpassung zu verhindern und die Generalisierungsfähigkeit des Modells zu verbessern. |
| **Batchgröße** | Gibt an, wie viele Trainingsbeispiele gleichzeitig in einem Optimierungsschritt verarbeitet werden. Eine größere Batchgröße stabilisiert die Gradienten, erfordert jedoch mehr Speicher. Kleinere Batches ermöglichen feinere Gewichtsanpassungen, können aber zu stärkerem Rauschen führen. |
| **Gradientenakkumulation** | Erlaubt das Aufsummieren von Gradienten über mehrere Mini-Batches, bevor ein Optimierungsschritt durchgeführt wird. Dadurch kann effektiv mit größeren Batchgrößen trainiert werden, ohne dass zusätzlicher GPU-Speicher benötigt wird. |
| **Warmup-Anteil** | Beschreibt den Prozentsatz der anfänglichen Trainingsschritte, in denen die Lernrate schrittweise von 0 auf den Zielwert erhöht wird. Dies stabilisiert das Training, insbesondere bei großen Modellen und kleinen Datensätzen. |
| **Optimierer** | Legt das Verfahren zur Gewichtsanpassung fest. Häufig verwendet wird *AdamW*, das Regularisierung mit adaptiven Lernraten kombiniert und für LLM-Fine-Tuning eine gute Balance zwischen Stabilität und Konvergenz bietet. |
| **Scheduler** | Bestimmt den zeitlichen Verlauf der Lernrate während des Trainings. Typische Ansätze sind *Cosine Decay* oder *Linear Decay*, die die Lernrate nach der Warmup-Phase schrittweise reduzieren, um ein sanftes Auslaufen des Trainings zu ermöglichen. |

Tabelle A 1: Übersicht der zentralen LoRA-Fine-Tuning-Parameter

Diese Parameter sind entscheidend für das effiziente domänenspezifische Fine-Tuning von LLMs mithilfe des PEFT-/LoRA-Ansatzes. Durch die gezielte Anpassung der *Target_modules* und Low-Rank-Matrizen (gesteuert über *r* und α) kann das Modell neue Aufgaben erlernen, ohne das gesamte Parametergefüge neu zu trainieren. *Lernrate*, *Epochenzahl*, *Dropout*, *Batchgröße* und *Gradientenakkumulation* beeinflussen maßgeblich die Trainingsstabilität, während *Warmup-Anteil*, *Optimierer* und *Scheduler* die Konvergenzgeschwindigkeit und den Glättungsverlauf der Lernrate steuern.

## 10.2 Optimierte Transformer-Architektur

In vielen modernen LLMs wird die Pre-Norm-Variante der Transformer-Architektur eingesetzt. Die Abbildung A 1 zeigt den Aufbau einer solchen optimierten Transformer-Architektur.

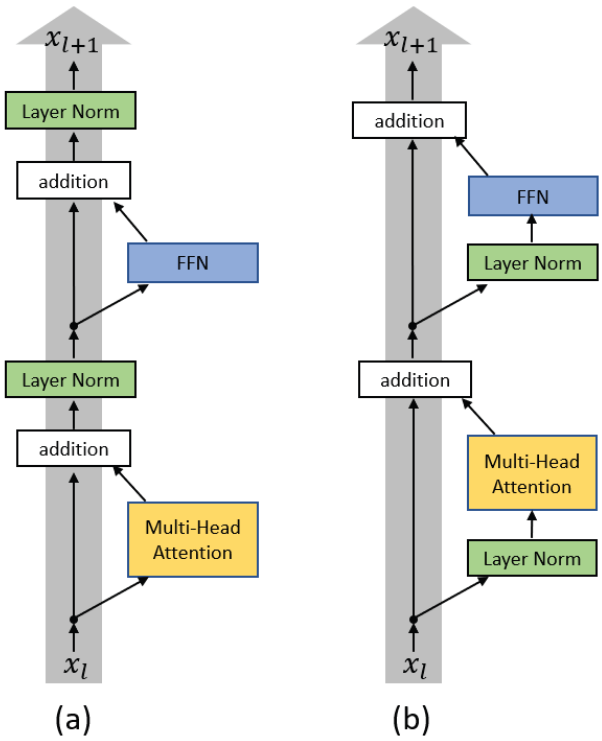


Abbildung A 1: Vergleich der Post-LN- und Pre-LN-Transformer-Architektur[XION20]

Im Vergleich zur ursprünglichen Architektur [VASW17] fällt auf, dass die Layer- Normalisierung vor die jeweilige Subschicht gestellt wird, anstatt danach. Diese Umstellung wirkt sich positiv auf die Trainingsstabilität bei tiefen Netzen aus und ermöglicht es, Modelle mit deutlich mehr Schichten erfolgreich zu trainieren. Weiterhin sind die Residual-Verbindungen klarer strukturiert. Die Eingabe eines Layers wird jeweils vor der Subschicht normalisiert, anschließend durch die Subschicht verarbeitet und dann mit der ursprünglichen Eingabe aufsummiert. Diese

Technik erleichtert den Gradientenfluss und wirkt dem Problem des Vanishing Gradient entgegen, bei dem die Gradienten in tiefen Netzen während des Backpropagationsprozesses immer kleiner werden und dadurch ein effektives Lernen der frühen Schichten verhindert wird. Im Decoder zeigt sich eine weitere Besonderheit. Die Masked Multi-Head Self-Attention verhindert nach wie vor, dass zukünftige Tokens sichtbar sind, allerdings wird die anschließende Cross-Attention explizit durch ein vorgeschaltetes Norm-Element stabilisiert. Durch diese Modifikationen können moderne Transformer-Varianten wie *GPT-3* robuster und skalierbarer trainiert werden. Diese optimierte Variante bildet die Grundlage für die meisten heutigen LLMs, da sie eine höhere Trainingsstabilität, schnellere Konvergenz und bessere Generalisierungseigenschaften bietet. [XION20]

## 10.3 XML-Struktur von TcPOU-Dateien

Das folgende Listing A 1 zeigt den strukturellen Aufbau einer typischen TwinCAT-POU-Datei im XML-Format. Es verdeutlicht die Trennung zwischen den Abschnitten Declaration (Deklaration der Variablen und Methoden) und Implementation (logische Umsetzung der Funktionslogik).

```
<?xml version="1.0" encoding="utf-8"?>
<TcPlcObject Version="1.1.0.1" ProductVersion="3.1.4026.13">
  <POU Name="FB_DeviceController" Id="{7cdc9c56-830b-4376-892a-5769c6e9ac3d}"
SpecialFunc="None">
    <Declaration><![CDATA[
        FUNCTION_BLOCK FB_DeviceController
        VAR
            bIsActive     : BOOL := FALSE;
            nSetpoint     : INT := 0;
            nActualValue  : INT := 0;
            tLastUpdate   : TIME;
            Start         : BOOL := FALSE;
        END_VAR
        ]]></Declaration>
    <Implementation>
        <ST><![CDATA[]]></ST>
    </Implementation>
    <Method Name="Start" Id="{151b3e61-a8cc-41b5-996f-b0b802998af1}">
        <Declaration><![CDATA[METHOD PUBLIC Start : BOOL
            VAR_INPUT
            END_VAR
            ]]></Declaration>
        <Implementation>
            <ST><![CDATA[bIsActive := TRUE;
            Start := TRUE;]]></ST></Implementation>
    </Method>
    <Method Name="Stop" Id="{09573a3b-5e74-4151-888e-d0b5d73e60f2}">
        <Declaration><![CDATA[METHOD PUBLIC Stop : BOOL
            VAR_INPUT
            END_VAR
            ]]></Declaration>
        <Implementation>
            <ST><![CDATA[bIsActive := FALSE;
            Start := FALSE;]]></ST></Implementation>
    </Method>
  </POU>
</TcPlcObject>
```

Listing A 1: Beispielhafte Darstellung einer TcPOU-Datei

Die Datei beschreibt den Funktionsbaustein FB_DeviceController, in dem die Variablen im Abschnitt *Declaration* deklariert und in den Methoden *Start* und *Stop* im Abschnitt *Implementation* logisch verarbeitet werden. Es handelt sich hierbei um ein kompaktes Beispiel, das den grundsätzlichen Aufbau einer TcPOU-Datei veranschaulicht. In realen Projekten können Funktionsblöcke, Programme oder Funktionen deutlich umfangreicher ausfallen. Der Umfang und

die Komplexität der XML-Dateien skalieren dabei in Abhängigkeit von der Anzahl der enthaltenen Variablen, Methoden und Implementierungslogiken, wodurch sowohl die Dateigröße als auch die Analysekomplexität zunimmt.

## 10.4 Weboberfläche zur Umsetzung des Tools

Die Abbildung A 2 zeigt den Funktionsbereich des Tools, in dem TwinCAT-Projekte oder -Dateien auf Strukturmerkmale, Richtlinien und Designprinzipien analysiert werden können. Außerdem kann auf der Seite die synthetische Datengenerierung auf Basis der Analyse gestartet werden.

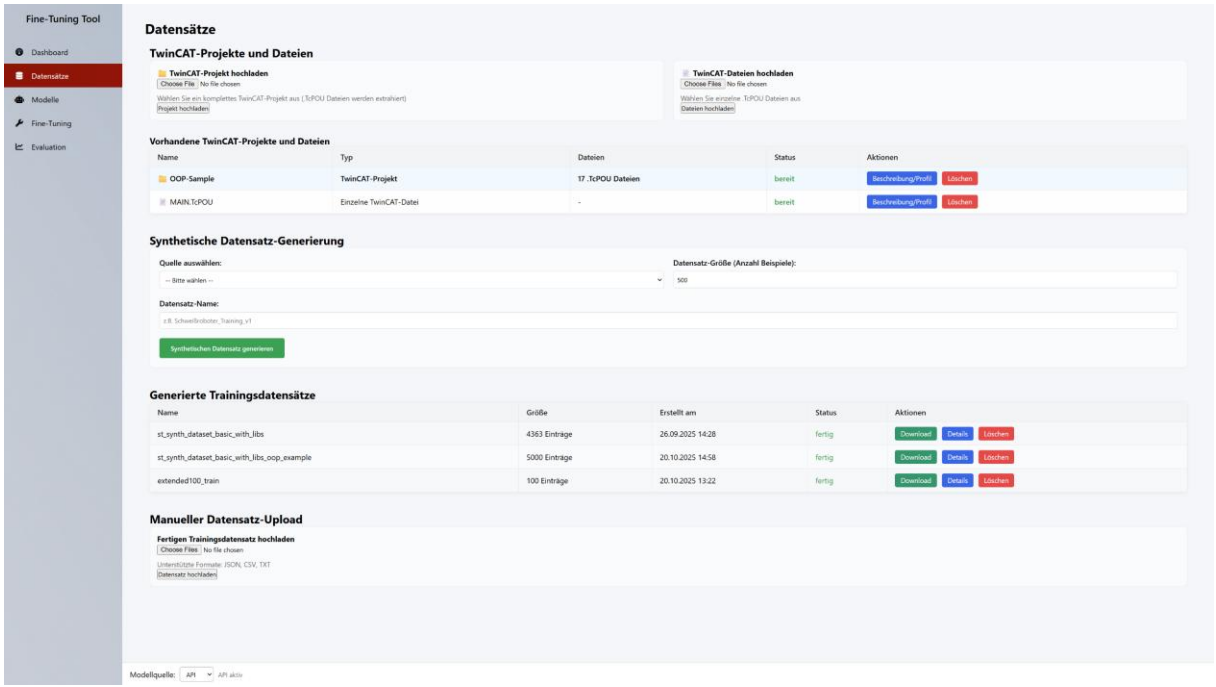


Abbildung A 2: Weboberfläche – Projektanalyse und synthetische Datengenerierung

Im oberen Abschnitt können vollständige TwinCAT-Projekte oder einzelne Dateien hochgeladen und automatisiert analysiert werden. Die daraus extrahierten Strukturinformationen bilden die Grundlage für die nachfolgende Generierung synthetischer Structured-Text-Daten. Die analysierten Projekte oder Dateien können nach der Analyse bei Bedarf spezifisch angepasst werden, oder gelöscht werden. Dadurch kann vor der Generierung von synthetischen Trainingsdaten sichergestellt werden, dass die Anforderungen an die Trainingsdaten den Erwartungen entsprechen. Die synthetische Datengenerierung erfolgt im mittleren Bereich, wobei die Analysedatei, Name und Umfang der zu erzeugende Trainingsdaten definiert werden kön-

nen. Im unteren Abschnitt werden die generierten Trainingsdatensätze verwalten. Die Trainingsdaten können als JSON-Datei heruntergeladen werden, im Detail analysiert werden und auch gelöscht werden. Alternativ befindet sich die Möglichkeit zum manuellen Hochladen von bereits vorhanden Trainingsdaten im unteren Bereich der Seite. In der Fußzeile der Seite kann der Analyse- und Generierungsmodus umgeschaltet werden. Es steht ein API-Modus zu Verfügung, mit dem über die Cloud-basierte API die Analyse und Generierung stattfindet. Alternativ kann auf den lokalen Modus gewechselt werden, damit die Projekte datenschutzgerecht verarbeitet werden können. Insgesamt stellt dieser Bereich die zentrale Schnittstelle zwischen der Projektanalyse und der Datensynthese im Gesamtsystem dar.

Die Abbildung A 3 zeigt einen Ausschnitt aus der Analysedatei. Auf dieser Seite werden die extrahierten Merkmale übersichtlich dargestellt.

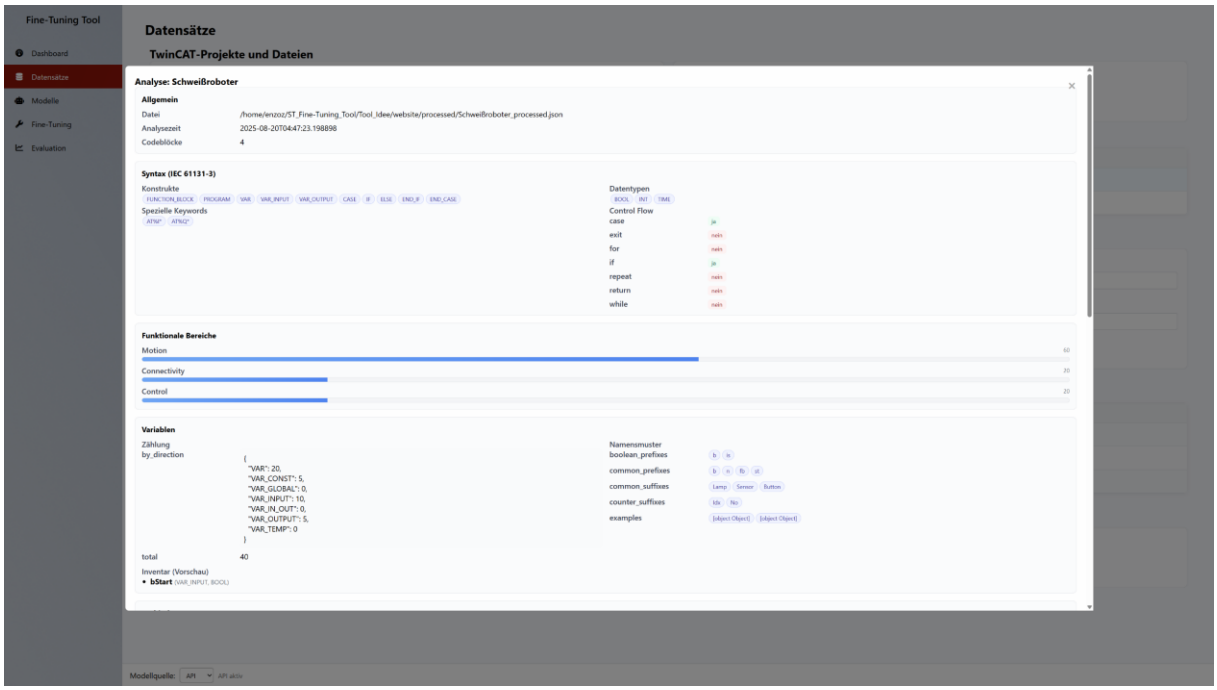


Abbildung A 3: Weboberfläche – Ausschnitt aus der Analysedatei

Nach dem Einlesen der Quellstruktur werden zentrale Merkmale extrahiert und nach IEC 61131-3 Kriterien klassifiziert. Die Analyse umfasst beispielsweise syntaktische Elemente wie Datentypen, Strukturen und Schlüsselwörter sowie funktionale Zuordnungen zu Funktionsbereichen. Zusätzlich werden Variablen und Namensmuster wie Präfixkonventionen erkannt, um wiederkehrende Strukturprinzipien zu identifizieren. Diese und weitere Metadaten dienen als Grundlage für die gezielte Generierung synthetischer Trainingsdaten.

Der Fine-Tuning-Prozess unterteilt sich in automatisiertes und manuelle Fine-Tuning. Die Abbildung A 4 zeigt die Weboberfläche für den automatisierten Fine-Tuning-Prozess.

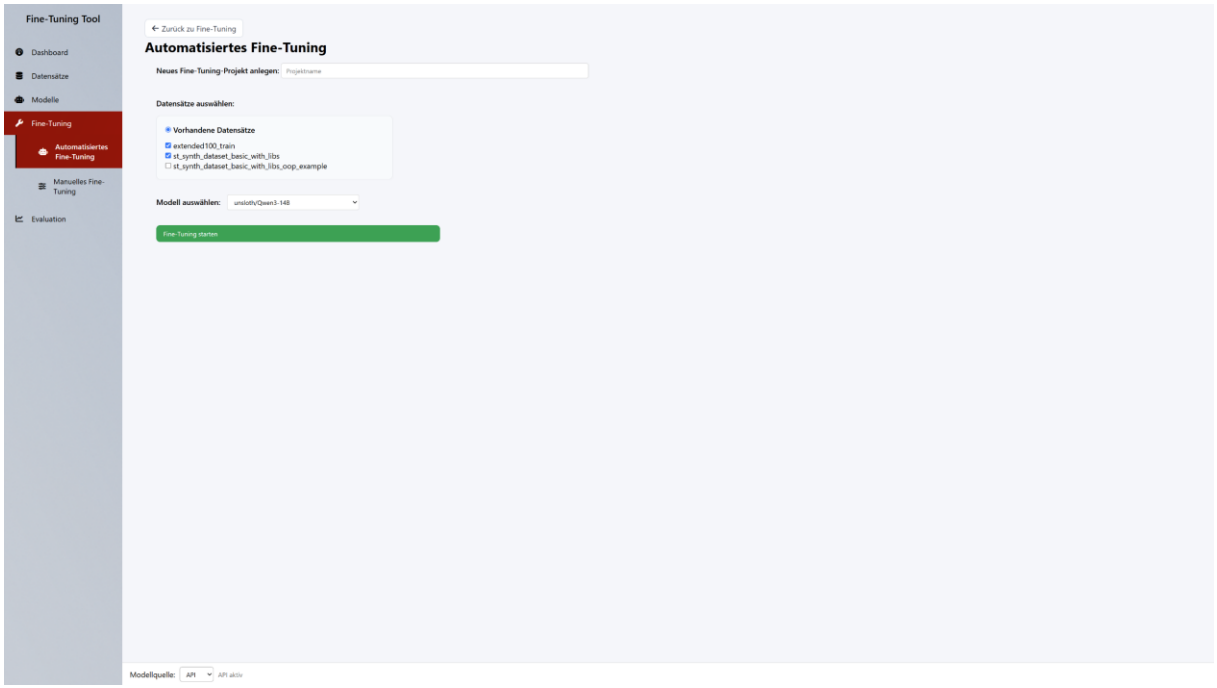


Abbildung A 4: Weboberfläche – Automatisiertes Fine-Tuning

Auf dieser Seite wählt der Nutzer einen zuvor synthetisch erzeugten oder importierten Trainingsdatensatz sowie ein Basismodell aus, auf das das Fine-Tuning angewendet werden soll. Man hat die Möglichkeit auch mehrere Datensätze für das Fine-Tuning auszuwählen, die dann vor dem Fine-Tuning zu einem gemeinsamen Datensatz verbunden werden. Zusätzlich muss für das Fine-Tuning ein Name definiert werden, um die angepassten LLMs nach dem Fine-Tuning identifizieren zu können. Die Parameter für das Fine-Tuning werden in diesem Modus automatisch gesetzt und auf Basis von Empfehlungen der LLM-Anbieter optimiert.

Falls die Parameter nicht automatisch gesetzt werden sollen, kann das durch die eigenständige Einstellung der Parameter vorgenommen werden. Die Abbildung A 5 zeigt die Seite der Weboberfläche, um die Parameter für das Fine-Tuning individuell festzulegen.

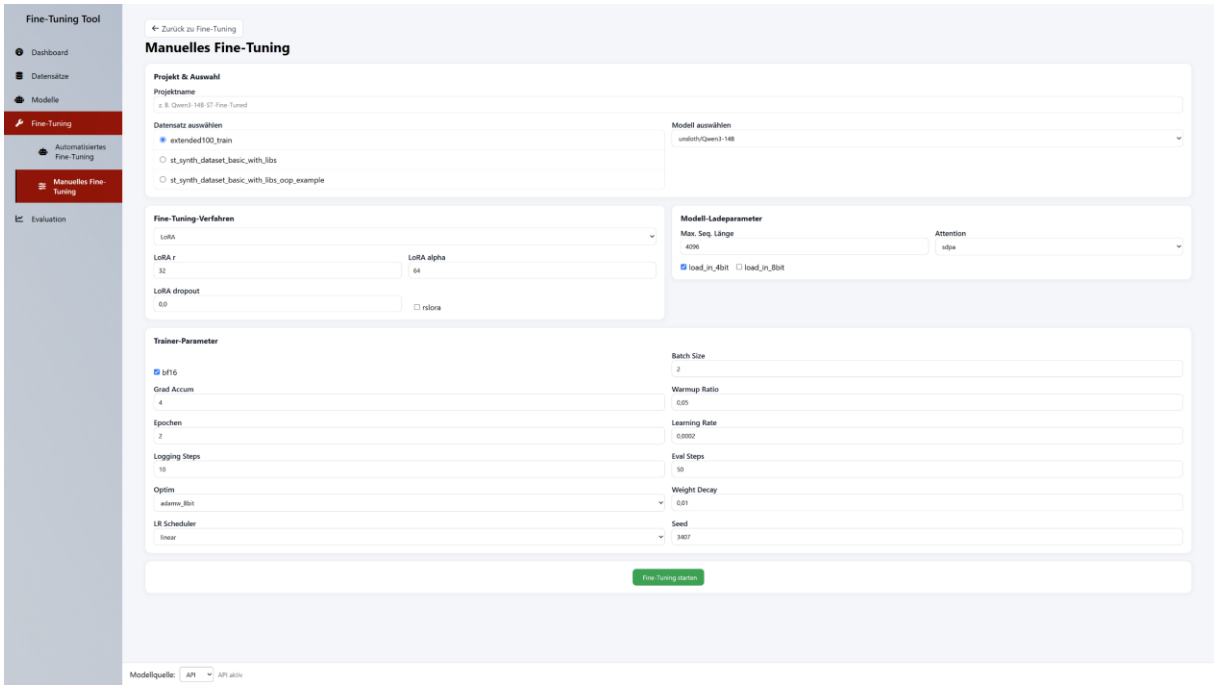


Abbildung A 5: Weboberfläche – Manuelles Fine-Tuning

Im Gegensatz zum automatisierten Prozess erlaubt dieser Bereiche eine gezielte Anpassung der LoRA-Hyperparameter (*r, α, Dropout*) sowie weiterer Trainingsgrößen wie Lernrate, Batchgröße und Epochenzahl. Darüber hinaus können weitere Modell-spezifische Parameter wie definiert werden. Diese Flexibilität ermöglicht es, verschiedene Fine-Tuning-Strategien zu erproben und deren Einfluss auf die syntaktische und semantische Modellqualität systematisch zu untersuchen. So dient das manuelle Fine-Tuning insbesondere der experimentellen Optimierung und dem Vergleich unterschiedlicher Trainingskonfigurationen innerhalb des Tools.

Nach dem Fine-Tuning stehen die angepassten LLMs auf der Modellseite der Weboberfläche zur Verfügung. Die Abbildung A 6 zeigt die Verwaltung von den LLMs nach dem Fine-Tuning über die Weboberfläche.

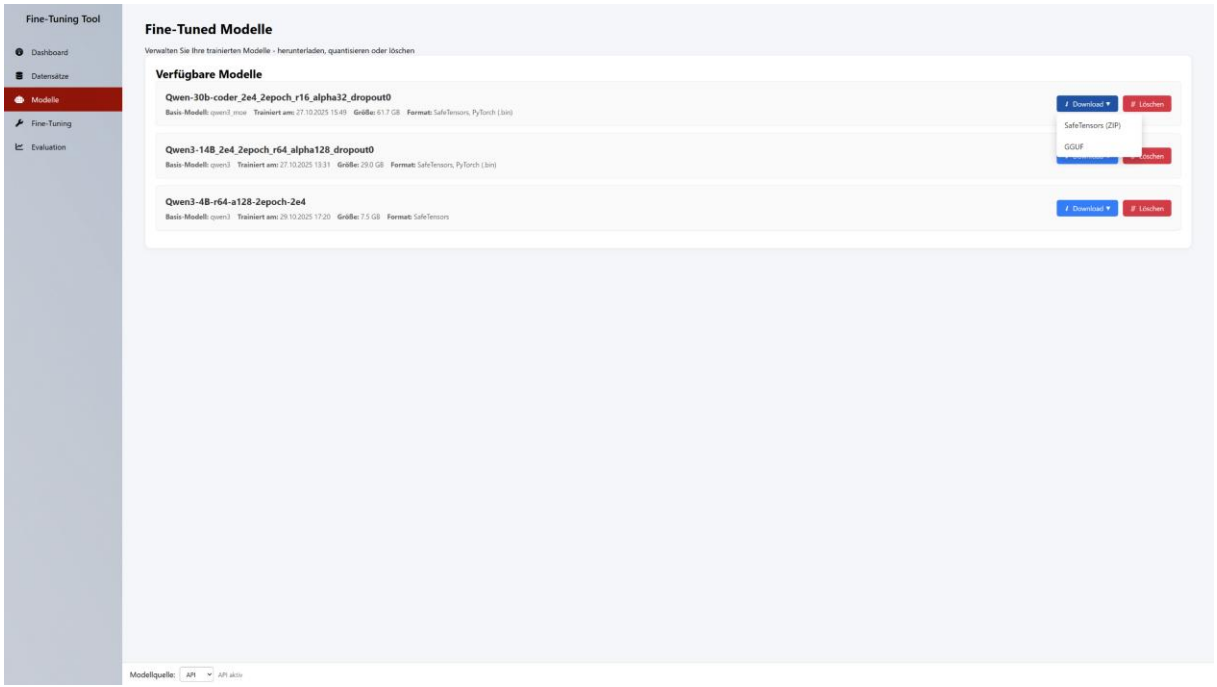


Abbildung A 6: Weboberfläche – Modellübersicht

Nach Abschluss des Fine-Tunings werden die resultierenden Modellvarianten auf dieser Seite zentral aufgelistet und mit ihren jeweiligen Trainingsnamen, Speichergrößen und Erstellungsdaten angezeigt. Neben der Übersicht über Modellvarianten können die Modelle in verschiedenen Formaten exportiert werden, etwa als *SafeTensors*- oder *GGUF*-Dateien. Diese Downloadmöglichkeiten ermöglichen die Weiterverwendung der angepassten Modelle innerhalb von Beckhoff-Systemen. Damit stellt dieser Bereich den Übergang von der Trainings- zur Anwendungsphase im Gesamtprozess des Tools dar.

Um die angepassten LLMs auf Qualität zu prüfen, stehen in dem Tool zwei Möglichkeiten zur Verfügung. Die Abbildung A 7 zeigt den Inferenzbereich des Tools, in dem LLM interaktiv getestet werden können.

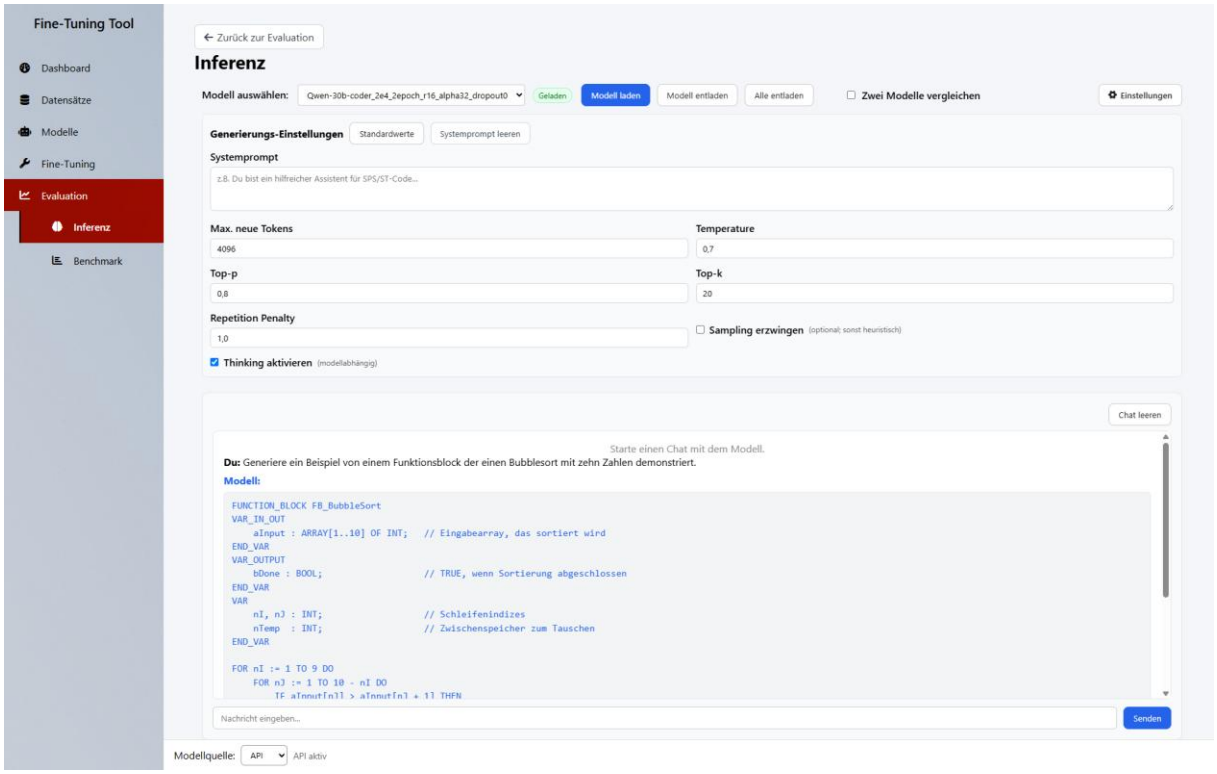


Abbildung A 7: Weboberfläche – Inferenzbereich

Der Nutzer hat die Möglichkeit verschiedene Basismodelle und angepasste Modelle zu laden, um die Ausgaben zu vergleichen. Nach Auswahl eines Modells lassen sich hier optional Generierungsparameter der Textgenerierung wie *Temperature*, *Top-k* oder *Repetition Penalty* anpassen, um das Ausgabeverhalten gezielt zu steuern. Werden diese nicht explizit verändert, werden die empfohlenen Generierungsparameter der LLM-Anbieter gesteuert. Im gezeigten Beispiel erzeugt das angepasste LLM *Qwen-30B-Coder* einen vollständigen Funktionsbaustein in Structured Text, der einen Bubblesort-Algorithmus implementiert. Diese Umgebung ermöglicht es, die syntaktische und semantische Qualität der Modellantworten direkt zu prüfen und dient zugleich als Grundlage für qualitative Vergleiche zwischen verschiedenen Fine-Tuning-Varianten innerhalb der Evaluationsphase.

Alternativ zu der Inferenz mit einem Modell zeigt die Abbildung A 8 den Vergleichsmodus der Inferenzoberfläche. Durch den Modus können zwei LLM-Varianten parallel gegenübergestellt werden.

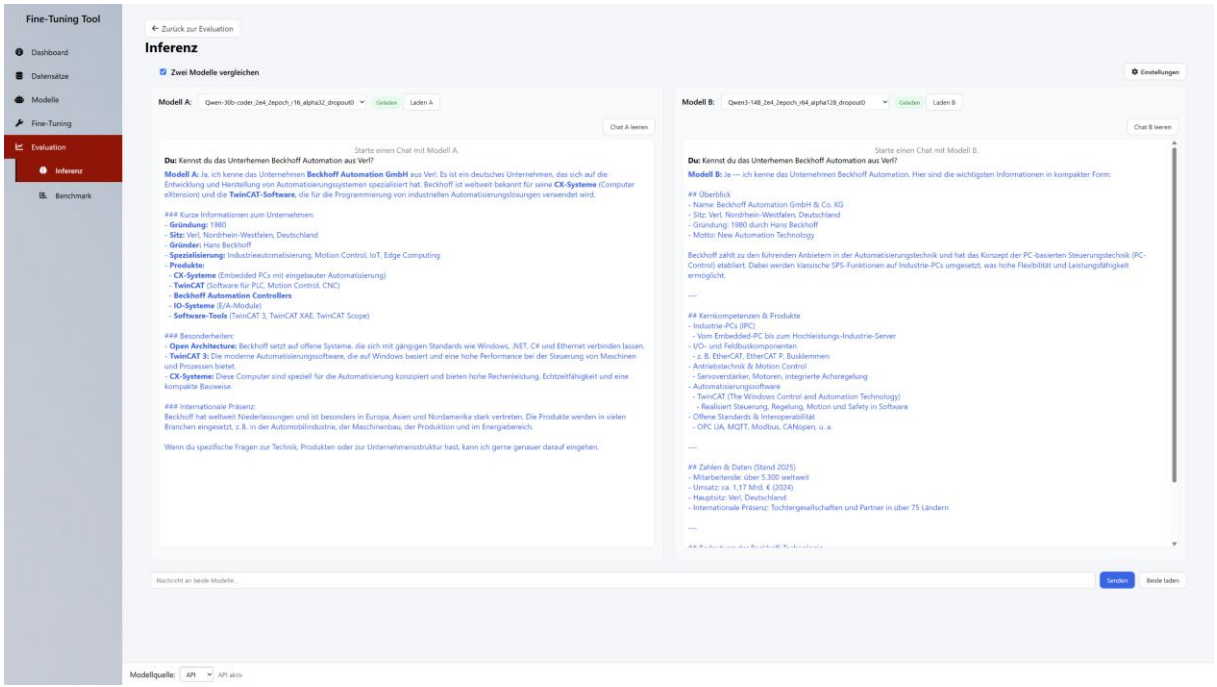


Abbildung A 8: Weboberfläche – Vergleichsmodus der Inferenzoberfläche

Durch die gleichzeitige Eingabe eines Prompts lassen sich die generierten Antworten der Modelle direkt vergleichen. Ziel dieser Funktion ist es, Unterschiede in Sprachstil, Informationsdichte und inhaltlicher Genauigkeit sichtbar zu machen und die Wirkung unterschiedlicher Fine-Tuning-Konfigurationen auf die Ausgabequalität zu bewerten. Zusätzlich kann in dieser Ansicht auch das jeweilige Basismodell der Fine-Tuning-Variante getestet werden, um die Auswirkungen des Fine-Tunings unmittelbar zu bewerten. Dadurch wird eine qualitative Einschätzung der Modelle im praktischen Anwendungskontext ermöglicht und die Auswahl der bestgeeigneten Variante für domänenspezifische Aufgaben unterstützt.

Für die objektive syntaktische und semantische Bewertung der LLMs existiert in dem Tool ein Benchmark- und Evaluationsbereich. Die Abbildung A 9 zeigt die Seite des Tools, um LLMs systematisch zu überprüfen.

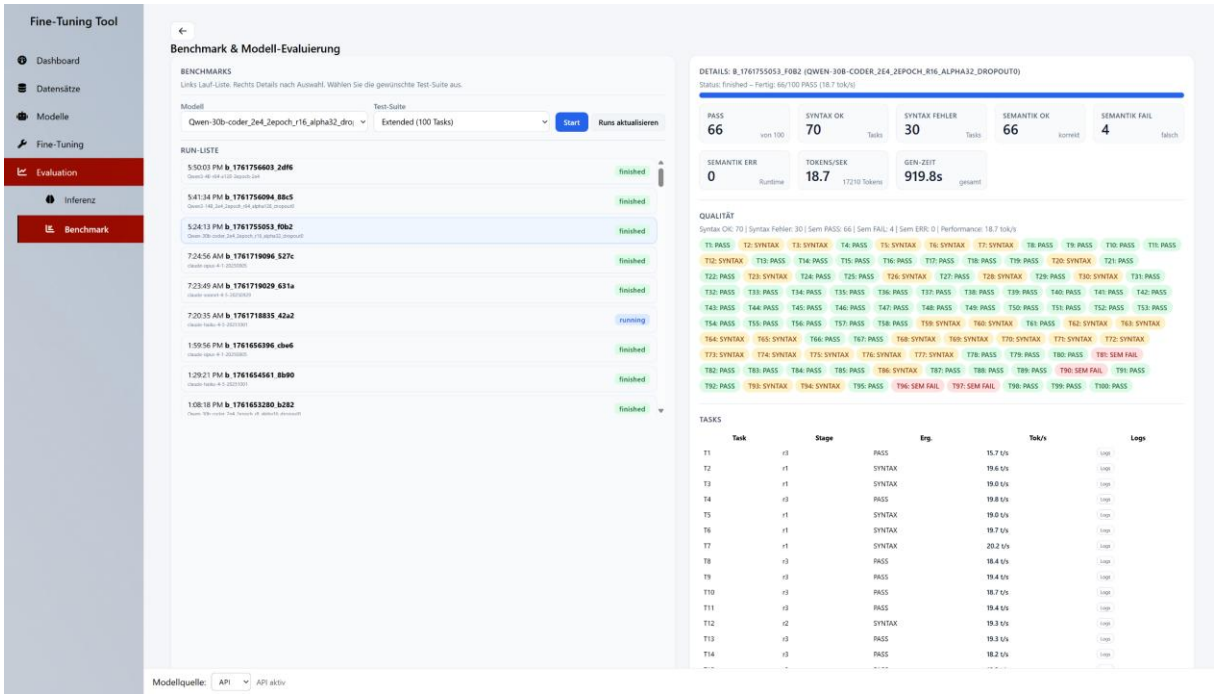


Abbildung A 9: Weboberfläche – Benchmark- und Evaluationsseite

Für den Benchmark wird eine definierte Test-Suite auf das LLM angewendet, um syntaktische und semantische Korrektheit objektiv zu bewerten. Die Ergebnisse werden in Kennzahlen wie zur Beschreibung der syntaktischen und semantischen Qualität zusammengefasst. Zusätzlich wird das Ergebnis jeder Aufgabe visuell dargestellt und in drei Kategorien eingeteilt (Pass, Syntaxfehler und Semantikfehler). Erweitern dazu kann man pro Aufgabe sich die Protokolle ansehen, um festzustellen, wo der Fehler bei der Generierung der Aufgabe vorliegt. Neben der syntaktischen und semantischen Qualität werden auch Metriken wie Generierungsrate und Generierungszeit erfasst. Diese Seite bildet den Abschluss des Fine-Tuning-Prozesses und ermöglicht eine direkte Vergleichbarkeit verschiedener Modellvarianten sowie die Identifikation der leistungsstärksten Konfigurationen.

## 10.5 Modellergebnisse

Im Folgenden werden die Ergebnisse der durchgeführten Fine-Tuning-Experimente für die verschiedenen Modellvarianten dargestellt. Für jede Modellfamilie wurde die syntaktische und semantische Qualität bewertet sowie die durchschnittliche Generierungsleistung in Tokens pro Sekunde ermittelt. Die jeweils besten Werte pro Modell sind in den Tabellen hervorgehoben.

Tabelle A 2 zeigt die Resultate des Fine-Tunings für das Modell *Qwen-4B*. Die Analyse umfasst die Erfassung der syntaktischen und semantischen Qualität sowie die Generierungsleistung in Tokens pro Sekunde.

| **Modell** | **Anzahl trainierter Parameter** | **Syntaktische Qualität** | **Semantische Qualität** | **Tokens / s** |
|---|---|---|---|---|
| **Basismodell** | / | 54 | 49 | **82,7** |
| **R8_A16** | 16.515.072 | 52 | 47 | 82,5 |
| **R16_32** | 33.030.144 | 54 | 48 | 82,5 |
| **R32_A64** | 66.060.288 | 62 | 50 | 82,5 |
| **R64_A128** | 132.120.576 | **67** | **54** | **82,7** |

Tabelle A 2: Fine-Tuning-Ergebnisse vom Qwen-4B

Mit zunehmendem Rang und LoRA-Alpha verbessert sich die syntaktische Qualität um 24 %, während die semantische Qualität um rund 10 % steigt. Die Generierungsgeschwindigkeit bleibt dabei nahezu konstant.

Tabelle A 3 fasst die Fine-Tuning-Ergebnisse für das *Qwen-14B*-Modell zusammen. Die Bewertung erfolgte anhand identischer Metriken zur Vergleichbarkeit der Ergebnisse.

| **Modell** | **Anzahl trainierter Parameter** | **Syntaktische Qualität** | **Semantische Qualität** | **Tokens / s** |
|---|---|---|---|---|
| **Basismodell** | / | 58 | 47 | **40,1** |
| **R8_A16** | 32.112.640 | 62 | 53 | 39,9 |
| **R16_32** | 64.225.280 | 67 | **59** | 39,9 |
| **R32_A64** | 128.450.560 | 67 | 57 | 40,0 |
| **R64_A128** | 256.901.120 | **71** | **59** | 40,0 |

Tabelle A 3: Fine-Tuning-Ergebnisse vom Qwen-14B

Das Fine-Tuning steigert die syntaktische Qualität von 58 auf 71 Punkte und die semantische von 47 auf 59 Punkte. Damit zeigt sich auch bei diesem Model eine deutliche Leistungsverbesserung bei stabiler Ausführungsgeschwindigkeit.

Tabelle A 4 stellt die Resultate des *Qwen-30B-Coder*-Modells dar, das für komplexe Codegenerierungen optimiert ist. Die Ergebnisse zeigen den Einfluss der Trainingsparameter auf die Modellqualität und Rechenleistung.

| **Modell** | **Anzahl trainierter Parameter** | **Syntaktische Qualität** | **Semantische Qualität** | **Tokens / s** |
|---|---|---|---|---|
| **Basismodell** | / | 68 | 59 | 18,1 |
| **R8_A16** | 421.920.768 | 58 | 52 | 18,6 |
| **R16_32** | 843.841.536 | **73** | **66** | **18,7** |
| **R32_A64** | 1.687.683.072 | 71 | 65 | **18,7** |
| **R64_A128** | 3.375.366.144 | 65 | 60 | **18,7** |

Tabelle A 4: Fine-Tuning-Ergebnisse vom Qwen-30B-Coder

Die Variante *R16_A32* liefert die beste Balance aus Qualität und Effizienz. Diese erreicht eine um 7 Punkte höhere semantische Qualität als das Basismodell bei unveränderter Inferenzgeschwindigkeit.

Tabelle A 5 zeigt die Trainingszeiten der optimalen Varianten der drei untersuchten Modellfamilien. Die Werte zeigen den Zusammenhang zwischen Anzahl an Parametern und Trainingsdauer.

| **Modell** | **Variante** | **Anzahl trainierter Parameter** | **Trainingsdauer** |
|---|---|---|---|
| **Qwen-4B** | R64_A128 | 132.120.576 | 8 Minuten und 35 Sekunden |
| **Qwen-14B** | R64_A128 | 256.901.120 | 21 Minuten und 6 Sekunden |
| **Qwen-30B-Coder** | R16_A32 | 843.841.536 | 330 Minuten und 50 Sekunden |

Tabelle A 5: Trainingszeiten der optimalen Trainingsvarianten

Die Werte basieren auf dem Fine-Tuning mit einem Datensatz von 5.000 Trainingsbeispielen, einer Epochenanzahl von zwei sowie einer Lernrate von 2e-4. Modelle mit einer geringeren

Anzahl an Parametern lassen sich in wenigen Minuten feinjustieren, während hochparametrisierte Modelle deutlich längere Trainingszeiten benötigen.

Tabelle A 6 enthält die Benchmark-Ergebnisse externer Anthropic-Modelle als Referenz. Diese dienen zur Einordnung der erzielten Fine-Tuning-Resultate.

| Modell | Syntaktische Qualität | Semantische Qualität | Tokens / s |
|---|---|---|---|
| **Opus-4-1-20250805** | **93** | **80** | 21,1 |
| **Sonnet-4-5-20250929** | 83 | 71 | 36,0 |
| **Haiku-4-5-20251001** | 81 | 75 | **64,3** |

Tabelle A 6: Benchmark-Ergebnisse von Anthropic-Modellen

Die Modelle von Anthropic erreichen mit syntaktischen Qualitätswerten bis 93 Punkten und semantischen bis 80 Punkten zwar die insgesamt besten Ergebnisse, was ihre überlegene Leistungsfähigkeit bei generellen Programmieraufgaben unterstreicht. Diese Resultate gehen jedoch mit laufenden Nutzungskosten pro Anfrage einher. Darüber hinaus ist ihr Einsatz aus Datenschutzsicht kritisch zu bewerten, da alle Berechnungen über externe Cloud-Server erfolgen und somit keine vollständige Datensicherheit gewährleistet ist.

Tabelle A 7 zeigt die Kostenkalkulation für die Generierung synthetischer Trainingsdaten. Grundlage bilden die Input- und Output-Token-Kosten pro Trainingseintrag.

| Modell | Input-Tokens | Output-Tokens | Input-Kosten | Output-Kosten | Kosten / Trainingseintrag |
|---|---|---|---|---|---|
| **Opus-4-1-20250805** | 801,0 | 113,9 | 15 $ / MTok | 75 $ / MTok | 0,0206 $ |
| **Sonnet-4-5-20250929** | 890,7 | 128,37 | 3 $ / MTok | 15 $ / MTok | 0,0046 $ |
| **Haiku-4-5-20251001** | 919,6 | 121,53 | 1 $ / MTok | 5 $ / MTok | 0,0015 $ |

Tabelle A 7: Kostenkalkulation für die synthetische Datengenerierung pro Trainingseintrag

Die Kalkulation zeigt deutliche Kostendifferenzen zwischen den Modellen. Effiziente Varianten wie Haiku ermöglichen die Erstellung großer synthetischer Datensätze zu minimalen Kosten

und eignen sich daher insbesondere für Trainingsszenarien, bei denen eine hohe Anzahl einfacher Beispiele benötigt wird. Soll hingegen komplexerer oder stark kontextabhängiger ST-Code trainiert werden, wie etwa Funktionsbausteine mit Zustandslogik oder Bibliotheksabhängigkeiten, sind leistungsfähigere Modelle wie *Sonnet 4.5* oder *Opus 4.1* besser geeignet. Somit ist stets abzuwägen, ob der Fokus auf der Quantität der Trainingsdaten oder auf der Tiefe und Komplexität der Beispiele liegt, um das jeweils optimale Kosten-Qualitäts-Verhältnis zu erreichen.

## 10.6 Generierungsbeispiel nach dem Fine-Tuning

Die Abbildung A 10 zeigt den direkten Vergleich der Codeausgabe eines nicht angepassten Basismodells (links) mit der eines angepassten Modells (rechts). Beide Modelle erhalten die Aufforderungen eine einfache Motoransteuerung als Beispielprogramm zu generieren.

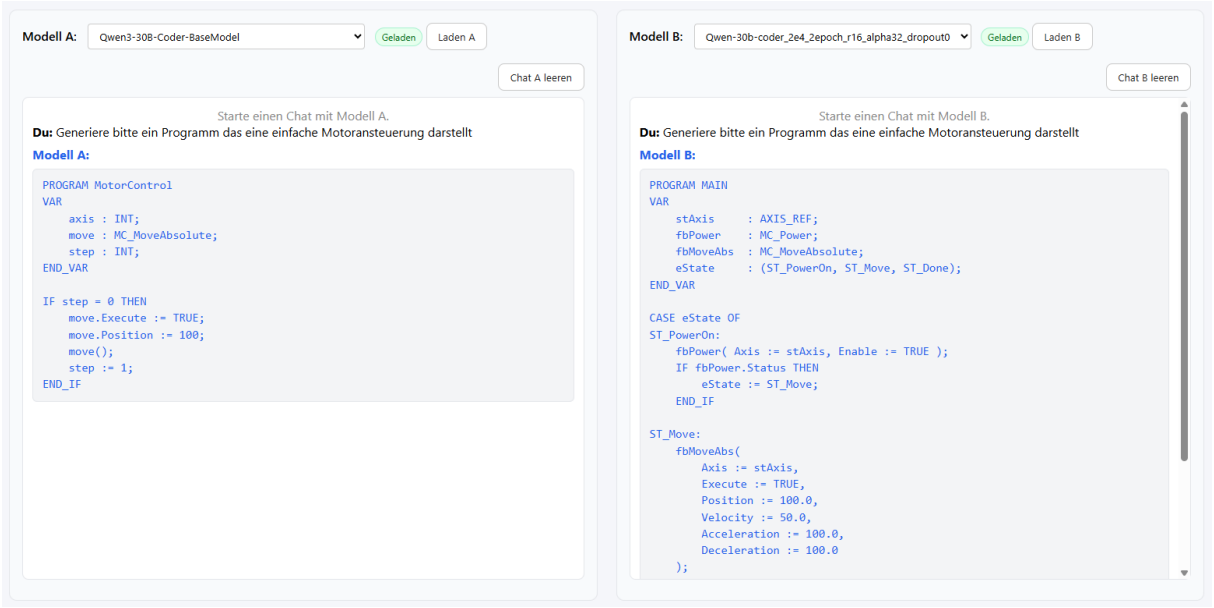


Abbildung A 10: Vergleich zwischen Basismodell und Fine-Tuned-Modell

Das Basismodell weist mehrere typische Generierungsfehler auf, wie eine unvollständige State-Machine-Struktur, fehlerhafte Aufrufe von Funktionsbausteinen sowie fehlende Achsenreferenzen. Das feinjustierte Modell hingegen berücksichtigt Namenskonventionen, verwendet die Beckhoff-Bibliothek *Tc2_MC2* korrekt und implementiert die Zustandslogik über eine strukturierte CASE-State-Machine. Der Vergleich verdeutlicht den Effekt des Fine-Tunings. Durch die Anpassung an domänenspezifische Trainingsdaten kann das LLM kundenspezifische Anforderungen besser abbilden und qualitativ hochwertigeren, lauffähigen ST-Code generieren.

HS'BI Hochschule Bielefeld University of Applied Sciences and Arts

10/2023

# I. Eigenständigkeitserklärung*

*Declaration of originality**

Hiermit versichere ich
*Hereby, I*

Zacharias, Enzo

Name, Vorname
*Last name, First name*

1321700

Matrikelnummer
*Student ID number*

Digitale Technologien

Studiengang
*Study programme*

dass ich die vorliegende Bachelorarbeit / bachelor thesis
*affirm that I have prepared the present*

(bei Gruppenarbeit mein bearbeiteter Teil) mit dem Thema
*(in case of group work the part I have prepared) with the topic*

Klicken oder tippen Sie hier, um Text einzugeben.

Entwicklung eines Tools zum automatisierten Fine-Tuning lokaler Large Language Models für kundenspezifische TwinCAT-Structured-Text-Anwendungen

selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen - einschließlich Tabellen, Karten, Abbildungen etc. -, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Werken und Quellen (dazu zählen auch Internetquellen) entnommen wurden, sind in jedem einzelnen Fall mit exakter Quellenangabe kenntlich gemacht worden.
*independently and without using any other than the indicated aids. All passages – including tables, maps, figures, etc. – taken verbatim or rephrased from published and unpublished works and sources (including Internet sources) have been identified in each individual case with exact reference to the source.*

Zusätzlich versichere ich, dass ich beim Einsatz von generativen IT-/KI-Werkzeugen (z. B. ChatGPT, BARD, Dall-E oder Stable Diffusion) diese Werkzeuge in einer Rubrik „Übersicht verwendeter Hilfsmittel" mit ihrem Produktnamen, der Zugriffsquelle (z. B. URL) und Angaben zu genutzten Funktionen der Software sowie Nutzungsumfang vollständig angeführt habe. Wörtliche sowie paraphrasierende Übernahmen aus Ergebnissen dieser Werkzeuge habe ich analog zu anderen Quellenangaben gekennzeichnet.
*In addition, I assure that, when using generative IT/AI tools (e.g., ChatGPT, BARD, Dall-E, Stable Diffusion), I have listed these tools in full in a section "Overview of tools used" with their product name, the access source (e.g., the URL) and information on the functions of the software used as well as the scope of use. I have marked verbatim and paraphrased quotes from the results of these tools in the same way as I have marked other sources.*

Mir ist bekannt, dass es sich bei einem Plagiat um eine Täuschung handelt, die gemäß der Prüfungsordnung sanktioniert werden wird.
*I am aware that plagiarism is a form of cheating that will be penalised according to the examination regulations.*

Ich versichere, dass ich die vorliegende Arbeit oder Teile daraus nicht bereits anderweitig innerhalb oder außerhalb der Hochschule als Prüfungsleistung eingereicht habe.

1

HS'BI Hochschule Bielefeld University of Applied Sciences and Arts

10/2023

*I certify that I have not already submitted the present work or parts thereof as an examination performance elsewhere within or outside the university.*

Verl, 17.11.2025

Ort, Datum
*Place, date*

Enzo Zacharias

Unterschrift
*Signature*

* Bitte legen Sie diese Eigenständigkeitserklärung ausgefüllt und unterzeichnet Ihrer Arbeit am Ende bei. Sollte diese fehlen, wird die Arbeit nicht korrigiert bzw. bei endgültiger Nichtvorlage als Täuschungsversuch gewertet.
** Please complete and sign this declaration of originality and enclose it with your work at the end. If this is missing, the work will not be evaluated or, in case of final non-submission, it will be considered an attempt to cheat.*