Hochschule Merseburg
University of Applied Sciences

Fachbereich Ingenieurs- und Naturwissenschaften
Studiengang Automatisierungstechnik und Informatik
(MAI)

Masterarbeit

Automatisches Erzeugen von Testfällen für Web-APIs mit Hilfe großer Sprachmodelle

Erstbetreuer Prof. Dr. Sven Karol
Zweitbetreuer Prof. Dr. Doreen Straß
Abgabedatum: 21.06.2024

eingereicht von
Robert Schönfeld
Matrikel: 25003

Merseburg, 18. Juni 2024

Abstract

Diese Masterarbeit zielt auf die Entwicklung eines automatisierten Verfahrens ab, das mithilfe bereits existierender großer Sprachmodelle (LLMs) Testfälle für Web-APIs generiert. Zu diesem Zweck wurde ein entsprechendes Konzept entwickelt und auf einer geeigneten Plattform implementiert. Im Rahmen der Entwicklung dieses Verfahrens wurde untersucht, inwieweit sich bestehende LLMs nutzen lassen, um Web-APIs zu testen, und es wurde dargelegt, wie LLM-basierte Verfahren zur Generierung von Testfällen in typische agile Softwareentwicklungsprozesse integriert werden können. Die vorgestellte Lösung wird zudem auch mit den Arbeiten von [Xia+23] und [Kis22] verglichen. Für die Entwicklung eines geeigneten Prompts, mit dem das LLM die notwendigen Funktionen generieren kann, wurde ein Optimierungsverfahren konzipiert. Die Ergebnisse der Optimierung wurden systematisch analysiert und verglichen, um die Prompts so anzupassen, dass das LLM möglichst zuverlässig korrekte Funktionen in Form von Generatoren zur Erzeugung von Testfällen generiert. Diese Generatoren wurden anschließend mit dem durch Swagger Codegen erzeugten API-Client kombiniert, um das eigentliche Test-Tool zu konstruieren. Die Ergebnisse zeigen grundsätzlich, dass LLMs zur Erzeugung von Testfällen verwendet werden können. Allerdings waren die dabei erzeugten Generatoren häufig fehlerhaft, was möglicherweise auf die Verwendung ungeeigneter Prompts zurückzuführen ist. Dennoch konnte ein Test-Tool erstellt werden, das in der Lage war, Anfragen an das System unter Test (SUT) zu senden und die Antworten entgegenzunehmen. Es fehlten jedoch wichtige Prozesse zur automatischen Dokumentation der Ergebnisse und zur Überprüfung auf mögliche Verstöße gegen die in der OpenAPI Specification (OAS) definierten Operationen.

Abstract

This master's thesis aims to develop an automated process that generates test cases for web APIs using existing large language models (LLMs). For this purpose, a corresponding concept was developed and implemented on a suitable platform. As part of the development of this procedure, the extent to which existing LLMs can be used to test web APIs was investigated and it was shown how LLM-based procedures for generating test cases can be integrated into typical agile software development processes. The solution presented is also compared with the work of [Xia+23] and [Kis22]. An optimization procedure was designed to develop a suitable prompt with which the LLM can generate the necessary functions. The results of the optimization were systematically analyzed and compared in order to adapt the prompts so that the LLM generates correct functions as reliably as possible in the form of generators for creating test cases. These generators were then combined with the API client created by Swagger Codegen to construct the actual test tool. The results basically show that LLMs can be used to generate test cases. However, the generators produced were often faulty, possibly due to the use of unsuitable prompts. Nevertheless, it was possible to create a test tool that was able to send requests to the system under test (SUT) and receive the responses. However, important processes for automatically documenting the results and checking for possible violations of the operations defined in the OpenAPI Specification (OAS) were missing.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

.....	.....
Ort, Datum	Unterschrift der Verfasserin/ des Verfassers

Inhaltsverzeichnis

1 Einleitung

Dieser Abschnitt dient als Einführung und verschafft einen einleitenden Überblick. Zu Beginn wird das behandelte Thema umrissen und der aktuelle Forschungsstand kurz skizziert. Anschließend werden die spezifischen Ziele der vorliegenden Arbeit präsentiert, die im weiteren Verlauf ausgearbeitet werden. Abschließend wird die Struktur der Arbeit erläutert, wobei die einzelnen Themenbereiche und deren Anordnung dargelegt werden.

1.1 Motivation

Web-Dienste stellen ihre Funktionen häufig über Representational State Transfer (REST)-Schnittstellen mittels Hypertext Transfer Protocol (HTTP) bereit, wodurch sie sich leicht in andere (Web-)Anwendungen integrieren lassen. Während die hinter den Schnittstellen liegenden Softwarekomponenten durch Unit-Tests [Ham04] verifiziert werden können, gehört die Funktionalität der Schnittstellen zur Integrationsebene und muss zusätzlich getestet werden. Neben der korrekten Bereitstellung der eigentlichen Funktionalität sind auch die Umsetzung der REST-Architektur und die korrekte Verwendung von HTTP-Statuscodes wichtige Aspekte. Die Anwendungsprogrammierschnittstelle (API) eines Web-Dienstes kann hierbei durch OpenAPI [Swa24], einen offenen Standard für REST-APIs, definiert werden.

Eine Möglichkeit, den Entwicklungsaufwand für das Testen von Web-APIs zu reduzieren, besteht im automatischen Generieren von Testfällen basierend auf einer gegebenen API-Spezifikation. Grundlegende Ansätze für das automatische Testen von Software und Programmen sind Fuzzing [DN84] und Property-Based Testing (PBT) [FB97]. Diese Ansätze zielen darauf ab, Testfälle mit einer möglichst hohen Testabdeckung oder zum Nachweis bestimmter erwünschter Programmeigenschaften vollständig automatisch zu erzeugen. Dabei kommen in der Regel konventionelle Verfahren wie lokale Optimierung oder genetische Algorithmen zum Einsatz. Neuere Arbeiten [Xia+23] zeigen jedoch auch, wie Künstliche Intelligenz (KI) für Fuzzing verwendet werden kann.

1.2 Zielsetzung

Diese Arbeit untersucht, inwieweit sich derzeit bestehende Large Language Models (LLMs) zur Generierung von Testfällen für das Testen von Web-APIs und Programmen nutzen lassen. Dabei wird auch betrachtet, ob die Integration von LLM-basierten Testansätzen in typischen agilen Softwareentwicklungsprozessen bereits praktikabel ist. Zu diesem Zweck wird ein geeignetes Konzept zur automatischen Erzeugung von Testfällen für Web-APIs unter Verwendung bestehender LLMs entwickelt. Dieses Konzept wird anschließend prototypisch auf einer geeigneten Hardware- oder Cloudplattform implementiert. Abschließend erfolgt eine Evaluation der Funktionalität und Praktikabilität des Prototyps, gefolgt von einem Vergleich mit bestehenden Lösungen.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in insgesamt fünf Abschnitte: Einleitung, theoretische Grundlagen, Konzeptentwicklung und Implementierung, Validierung sowie Fazit und Ausblick. Im ersten Teil wird die Motivation des Themas beschrieben und die Zielsetzung der Masterarbeit konkretisiert. Den Schluss macht der hier zu lesende inhaltliche Rahmen. Die theoretschen Grundlagen umfassen die Kapitel 2 und 3. Zunächst werden in Kapitel 2 die Prinzipien von Random Testing im zusammenhang mit PBT erläutert. Anschließend werden in Kapitel 3 wichtige Grundlagen im Themenbereich der LLMs detailiert betrachtet. Dabei wird zum einen die Architektur „Attention is all you need“ [Vas+17], die eine zentrale Grundlage für viele entwickelte Modelle darstellt, behandelt, als auch Verfahren zur Optimierung der Modellausgaben vorgestellt. Währenddessen werden auch State-of-the-Art Applikationen vorgestellt, die im Bereich der Softwaretests verwendet werden. Der Abschnitt zur Konzeptentwicklung und Implementierung umfasst die beiden Kapitel 4 und 5. In Kapitel 4 wird einerseits die OpenAPI-Spezifikation (OAS) betrachtet, um mögliche Untersuchungsziele sowie erforderliche Einschränkungen zu spezifizieren. Andererseits werden Anforderungen festgelegt, die von den LLMs erfüllt werden müssen, um die Auswahl auf möglichst wenige, aber vielversprechende Modelle einzuschränken. Zudem werden in diesem Kapitel eine Struktur für die Prompts sowie ein geeignetes Verfahren zur Evaluierung der Modelle entwickelt. Des Weiteren werden verschiedene Dekodierstrategien vorgestellt, die später bei der Evaluierung mit einbezogen werden. In Kapitel 5 wird schließlich die konkrete Implementierung des ausgewählten Konzepts in seinen einzelnen Phasen detailliert erläutert. Die Kapitel 6, 7 und 8 bilden den Abschnitt

zur Validierung. In Kapitel 6 werden zunächst die verschiedenen Testumgebungen und deren Verwendung beschrieben. Anschließend folgen Erläuterungen zu den Vorbereitungen, wie dem Aufsetzen der Web-API, sowie den für die Evaluierung ausgewählten LLMs. Abgeschlossen wird das Kapitel mit der Beschreibung der Vorgehensweise bei den durchgeführten Tests. In Kapitel 7 werden die gewonnenen Ergebnisse zusammengefasst und in Kapitel 8 werden diese schließlich Diskutiert. Der letzte Abschnitt besteht aus den Kapiteln 9 und 10. In diesen Kapiteln werden die Ergebnisse und erreichten Ziele in Form eines Fazits zusammengefasst. Abschließend wird noch ein Ausblick auf mögliche zukünftige Entwicklungen und Forschungsansätze vorgestellt.

2 Testen von Programmen

In diesem Kapitel werden zunächst Random Testing und die damit verbundenen Begriffe erläutert. Anschließend wird der Zusammenhang sowie die Anwendung von PBT und Fuzzing beschrieben. Hierbei werden auch bekannte Probleme dieser Testmethodiken hervorgehoben und betrachtet. Abschließend wird noch erläutert, wie die Ergebnisse von PBT und Fuzzing zu interpretieren sind.

2.1 Random Testing

Begriffsdefinitionen

Zunächst werden einige für diese Arbeit relevante Begriffe definiert. Zur Veranschaulichung wird das folgende Beispiel einer in Python geschriebenen Funktion betrachtet:

```python
def mul(num1: int, num2: int) -> int:
  return num1 * num2
```

Dabei ist «`mul`» die Funktion, die vom System unter Test (SUT) bereitgestellt wird. Das SUT ist ein System, das diese und möglicherweise auch weitere Funktionen über eine API bereitstellt, welches getestet werden soll. Die Variablen «`num1`» und «`num2`» sind die beiden Eingabeparameter, mit denen die Funktion aufgerufen wird. Ein Wert bezeichnet eine konkrete Instanziierung dieser Parameter, wie beispielsweise die Werte 2 und 5. Nach [FB97] besteht ein Test aus einer Reihe von Ausführungen eines bestimmten Programms, wobei bei jeder Ausführung unterschiedliche Eingabeparameter verwendet werden. Softwaretests umfassen alle Aktivitäten, die darauf abzielen, ein Attribut oder eine Fähigkeit eines Programms oder Systems zu bewerten und festzustellen, ob es die geforderten Ergebnisse erzielt [Pan99].

Komplexität von Softwaretests

Die Schwierigkeit beim Testen ergibt sich aus der Komplexität der Software. So kann z. B. ein Programm mit mäßiger Komplexität nicht vollständig getestet werden [Pan99; FB97]. Hierzu soll das im Beispiel dargestellte Programm zur Multiplikation zweier Zahlen erneut betrachtet werden, wobei angenommen wird, dass sich der Wertebereich im 64-Bit-Bereich befindet. Da dies 2^{128} verschiedene Testfälle ergibt, würde es selbst bei einer Testdurchführungsrate von tausend Operationen pro Sekunde eine beträchtliche Zeit in Anspruch nehmen, um alle möglichen Fälle zu testen. Dies trifft jedoch nur auf sogenannte „Batch“-Programme zu.

Bei interaktiven Programmen, die auch Eingaben aus der realen Welt einbeziehen, verschärft sich das Problem, da zeitliche und menschliche Interaktionen zusätzlich als mögliche Eingabeparameter berücksichtigt werden müssen [Pan99; CH00]. Eine weitere Komplikation ergibt sich aus der dynamischen Natur von Programmen. Wenn während der Vorabtests ein Fehler auftritt und der Code geändert wird, kann die Software nun für einen Testfall funktionieren, für den sie zuvor nicht funktionierte. Jedoch kann das Verhalten bei Testfällen, die vor dem Fehler bestanden wurden, nicht mehr garantiert werden [Pan99]. Da es schlichtweg nicht möglich ist, ein Programm auf alle möglichen Szenarien zu überprüfen, stehen verschiedene Verfahren zur Verfügung, sowohl systematische als auch zufällige, mit denen ein breites Spektrum abgedeckt werden kann. Es besteht jedoch immer noch ein gewisses Risiko, dass falsche Validierungen erzeugt werden können [FB97]. Wie in [Gol+24; Pan99] ausgeführt wird, trägt die Durchführung solcher Tests dennoch wesentlich zur Sicherstellung der korrekten Funktionsweise eines Programms bei. Darüber hinaus ermöglichen sie Aussagen zur Qualitätssicherung, Verifizierung und Validierung. Die meisten systematischen Testmethoden sind in der Praxis aus der Idee der „Abdeckung“ entstanden. Zufälliges Testen hingegen beansprucht keine Abdeckung [Ham94]. Studien haben dennoch gezeigt, dass das zufällige Testen von Software mit systematischen Verfahren in Bezug auf die Fehlerfindung konkurrieren kann [DN84; HT90].

Nutzen von Softwaretests

Unter der Annahme, dass die oben gezeigte Multiplikationsfunktion vom SUT bereitgestellt wird, kann das Verfahren des Random Testing gemäß [Ham94] wie folgt beschrieben werden: Zuerst werden alle Funktionen im SUT identifiziert, einschließlich ihrer Eingabebereiche. In diesem Fall wäre das nur die Funktion «`mul(int, int)`» mit zwei Eingabeparametern im Integer-Bereich. Im nächsten Schritt wird für den Testvorgang eine Menge an Testfällen N festgelegt. Die Wahl von N wird meistens auf Basis der verfügbaren Rechenleistung gewählt. Aber der Wert bestimmt am Ende auch darüber, wie Zuverlässig das Ergebniss des Testvorgangs ist. Ein Pseudozufallsgenerator erzeugt anschließend die notwendigen Werte für die Testfälle. Das zu testende Programm wird dann anschließend mit diesen generierten Eingaben getestet. Dabei schlägt ein Test fehl, wenn eine Eingabe zu falschen Ergebnissen führt. Andernfalls ist dieser erfolgreich. Um festzustellen, ob ein Ergebnis falsch ist, muss ein sogenanntes Orakel einbezogen werden [CH00]. Das Orakel beschreibt das erwartete Verhalten eines Systems. Ohne die Beurteilung von Testergebnissen durch formale Testorakel kann das Ziel des Testens, Fehler aufzudecken oder korrektes Verhalten zu gewährleisten, nicht praktisch erreicht werden. Eine manuelle Überprüfung der Ergebnisse ist weder zuverlässig noch kosteneffizient [RAO92]. Werden Fehler festgestellt, wird

Random Testing

das Programm korrigiert und der Test mit einem neuen Satz von Pseudozufallsdaten wiederholt.

Beurteilung der Zuverlässigkeit

Um eine Aussage über die Zuverlässigkeit eines Programms zu treffen, muss, wie in [DN84] erläutert wird, eine Ausfallrate ermittelt werden. Die Ausfallrate ist eine Kenngröße für die Zuverlässigkeit und entspricht in diesem Fall der Wahrscheinlichkeit, dass ein Fehler bei der Ausführung eines Programms zu einem bestimmten Zeitpunkt auftritt. Diese Wahrscheinlichkeiten zur Ausfallrate gelten jedoch nur in Bezug auf die für die Auswahl der Testdaten verwendete Eingabeverteilung. Anders ausgedrückt, die Ausfallrate gilt nur für das aktuell verwendete Betriebsprofil. Die Ausfallrate θ (Theta) eines „Batch"-Programms kann berechnet werden, indem die Anzahl der gefundenen Fehler F durch die Anzahl der durchgeführten Tests N geteilt wird:

$$\theta = \frac{F}{N}$$

Wenn beispielsweise 10 Fehler in 1000 durchgeführten Tests gefunden werden, beträgt die Ausfallrate 0,01. Im Falle eines interaktiven Programms entspricht N dem Zeitraum, in dem die Tests durchgeführt werden. Um zu beschreiben mit welcher Wahrscheinlichkeit bei N Tests bis zu F Fehler auftreten, wird die folgende kumulative Verteilungsfunktion verwendet [DN84]:

$$\sum_{i=1}^{F} \binom{N}{i} \theta^i \left(1-\theta\right)^{N-i} > \alpha$$

Das Ziel dieser Gleichung ist es, die obere Grenze für die Fehlerwahrscheinlichkeit θ' zu bestimmen, sodass der Wert von α nicht überschritten wird. Das Signifikanzniveau α wird oft vor Beginn des Experiments festgelegt und typischerweise auf einen bestimmten Wert wie 0,05 oder 0,01 gesetzt. Ein α-Wert von 0,05 bedeutet beispielsweise, dass eine 5%ige Wahrscheinlichkeit existiert, dass der Test einen Fehler identifiziert, wenn in Wirklichkeit kein Fehler vorliegt (Typ-I-Fehler). Ein niedrigerer α-Wert bedeutet ein geringeres Risiko, einen Typ-I-Fehler zu begehen, aber auch ein höheres Anspruchsniveau an die Evidenz, um eine Hypothese abzulehnen. Um θ' zu bestimmen, wird die kumulative Verteilungsfunktion gleich α gesetzt und nach nach θ aufgelöst. Dies ermöglicht es uns, die maximale Ausfallrate zu finden, die akzeptabel ist, um das gewünschte Signifikanzniveau α nicht zu überschreiten. Zudem kann es auch vorkommen, dass während eines Tests keine Fehler entdeckt werden. In diesem Fall wird die folgende Formel für

die Ausfallrate verwendet [DN84]:

$$\theta' = 1 - \alpha^{\frac{1}{N}}$$

Die Erfolgswahrscheinlichkeit, dass ein einziger Testfall erfolgreich ist, kann mit $1 - \theta$ berechnet werden. Bei N Testfällen wird dann die universellen Erfolgswahrscheinlichkeit eines Testvorgangs schließlich mit $(1 - \theta)^N$ berechnet. Mit dieser Erkenntnis lässt sich dann auch die Wahrscheinlichkeit, dass man mindestens einen Fehler in N Testfällen findet, wie folgt berechnen:

$$P = 1 - (1 - \theta)^N$$

An dieser Stelle soll nochmals betont werden, dass diese Wahrscheinlichkeiten stets nur für das Betriebsprofil gelten, für das die Tests durchgeführt werden. Zudem ist es nicht möglich, eine exakte Abbildung der Einsatzumgebung des SUTs zu erzeugen. In [Ham94] wird daher betont, dass diese Berechnungen sich dazu eignen, um verschiedene Methoden innerhalb der Forschung miteinander zu vergleichen.

2.2 Property-based Testing

Wie bereits in Kapitel 2.1 erwähnt, liegt die Stärke des Random Testings nicht in der Erreichung einer möglichst großen Abdeckung, sondern darin, durch die Generierung einer großen Menge an Testfällen möglichst viele Fehler aufzudecken und Aussagen über das zukünftige Verhalten der Software zu treffen [Ham94]. Wie in [CH00] erläutert, ermöglicht die Automatisierung schnelleres und gründlicheres Testen innerhalb der verfügbaren Zeit. Darüber hinaus erleichtert es die Wiederholung der Tests nach jeder Programmanpassung, was die Notwendigkeit einer automatischen Validierung der Ausgaben unverzichtbar macht [CH00]. Um eine solche Validierung zu ermöglichen, wird das Verfahren des PBT angewendet, welches in diesem Kapitel vorgestellt wird.

Property-Based Testing

PBT ist eine Testmethodik, bei der Tester ausführbare formale Spezifikationen, auch bekannt als Properties, von Softwarekomponenten erstellen. Ein automatisiertes System überprüft diese Spezifikationen anhand zahlreicher automatisch generierter Eingaben und Ausgaben [Gol+24]. Anstelle von Tests, die Beispiele für ein einzelnes konkretes Verhalten liefern, spezifizieren diese Tests Properties, die für eine breite Palette von Eingaben gelten. Die Testbibliotheken versuchen dann, Testfälle zu generieren, um diese Properties zu widerlegen [MHC19].

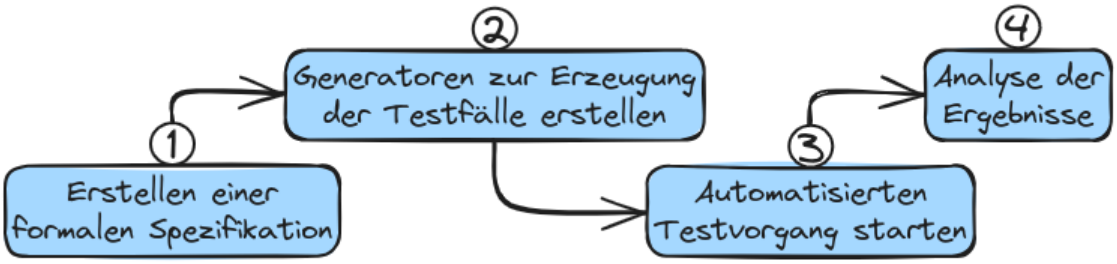


Abbildung 1: Prozessablauf von PBT

QuickCheck ist hierbei eines der ersten Werkzeuge, das Haskell-Programmierern beim Formulieren und Testen von Programmeigenschaften unterstützt [CH00] und 2006 erstmalig in einen bestehenden Softwaretest-Workflow integriert wurde [Art+06].

Prozessablauf Phase I & II

Die Vorgehensweise von PBT kann in mehreren Arbeitsschritten betrachtet werden, wie in Abbildung 1 dargestellt. Vorausgesetzt, der Tester verfügt über ein bestehendes Programm, das getestet werden muss, so muss der Tester zunächst einen geeigneten Standard zur Beschreibung der Properties auswählen. Falls das zu testende Programm eine Web-API ist, die über das SUT bereitgestellt wird, kann für die Beschreibung der Properties die OAS verwendet werden. Im nächsten Schritt ist es erforderlich, dass der Tester sämtliche definierbaren Properties identifiziert und diese manuell in der Spezifikation dokumentiert. Bezogen auf das Beispiel, das in Kapitel 2.1 verwendet wird, würde die Spezifikation lediglich ein einzelnes Property umfassen. Eine dazu passende OAS könnte dann wie in Listing 1 aussehen.

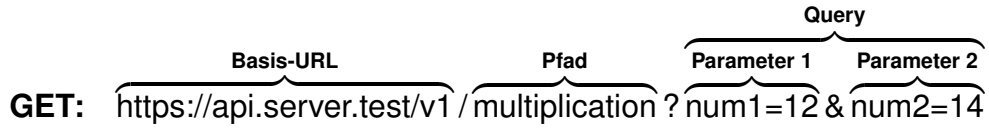
Erklärung zur OAS

Die OAS, auch bekannt als Swagger, ist ein Standard zur vollständigen Beschreibung von REST APIs [Swa24]. Von Zeile 1 bis 4 werden die Metadaten beschrieben, während Zeile 5 und 6 eine Liste von Basis-URLs für die Server enthält, wodurch beispielsweise URLs für Produktions- und Testserver gleichzeitig festgelegt werden können. Ab Zeile 7 werden die `«paths»` definiert, also alle möglichen Endpunkte, die von der API bereitgestellt werden. Diese Endpunkte sind relativ zur Server-URL. Im Beispiel wird der Endpunkt `«/multiplication»` (Zeile 8) mit der HTTP-Methode `«get»` (Zeile 9) definiert. Mit `«parameters»` (Zeile 10) wird eine Liste an Eingabeparameter beschrieben. Mit dem Schlüsselwort `«in»` (Zeilen 11 und 17) wird festgelegt, ob es sich um einen *path*-, *query*-, *header*- oder *cookie*-Parameter handelt, während `«name»` (Zeilen 12 und 18) den Variablennamen festgelegt. Im `«schema»` (Zeile 13 und 19) wird der Eingabeparameter weiter spezifiziert.

```yaml
openapi: '3.0.2'
info:
  title: API Title
  version: '1.0'
servers:
  - url: https://api.server.test/v1
paths:
  /multiplication:
    get:
      parameters:
        - in: query
          name: num1
          schema:
            type: integer
            minimum: 0
          required: true
        - in: query
          name: num2
          schema:
            type: integer
            minimum: 0
          required: true
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: object
                properties:
                  result:
                    type: integer
```

Listing 1: Beispiel einer OpenAPI Spezifikation

Die URL zur Erzeugung einer Anfrage könnte dann wie Folgt aussehen:



Ab Zeile 23 werden schließlich alle möglichen «responses» definiert, die vom Server generiert werden können. In diesem Beispiel würde der Server bei einer erfolgreichen Anfrage eine Antwort in Form einer JSON-Datenstruktur erzeugen, die das Ergebnis 168 enthalten sollte.

Prozessablauf Phase III

Gemäß [CH00] sollte ein Test-Tool in der Lage sein, Testfälle automatisch zu generieren. Dafür ist es üblicherweise erforderlich, vor dem Start des automatisierten Testvorgangs Generatoren zur Erzeugung von Eingabeparametern für die einzelnen Testfälle zu konstruieren. Bei Tools wie QuickCheck werden be-

reits vorab entwickelte Funktionen für die Erzeugung zufälliger Werte für einfache Datentypen über die Bibliothek bereitgestellt [CH00]. Jedoch genügen diese in den meisten Fällen nicht. In einer Befragung geht hervor, dass TesterInnen oft „[. . .] struggling to generate distributions of test examples that they were convinced effectively exercised the property, and sometimes viewed the process of designing random data generators as a distraction“ [Gol+24]. Dies liegt oft daran, dass Properties Vorbedingungen haben, die definieren, was es bedeutet, dass eine Eingabe gültig ist. Die Vorbedingung für die beiden Eingabeparameter gemäß dem vorliegenden Beispiel bestimmt, dass nur ganzzahlige Werte, die größer oder gleich Null sind, als gültig betrachtet werden. Ein anderes weniger triviales Beispiel einer Vorbedingung wäre die Vergabe einer Lager ID. Die ID ist hierbei ein String-Typ, der aus mehreren Teilen (z. B. „02-a-134-ar45“) besteht und mit einem „-“ getrennt wird. Dabei hat jeder einzelne Teil seine eigenen Vorbedingungen. Aufgrund solcher anspruchsvollen Vorbedingungen ist es oft notwendig, dass die Generatoren handgeschrieben sein müssen, um ausschließlich valide Eingabewerte zu erzeugen.

Prozessablauf Phase IV

Wenn eine formale Spezifikation mit den zu beschreibenden Properties vorliegt und die notwendigen Generatoren für die Erzeugung der Eingabeparameter existieren, kann der automatisierte Testvorgang gestartet werden. Unter Verwendung der Generatoren wird für jeden definierten Endpunkt (Property) eine Liste mit zufälligen Eingabewerten erzeugt. Die Gesamtheit aller Listeneinträge (Testfälle) bildet schließlich das Testset. Die Testfälle werden in einer zufälligen Reihenfolge ausgeführt. Wie in [RAO92] erläutert wird, kann das Systemverhalten anschließend im Hinblick auf eine Verhaltensspezifikation überprüft werden, um die Ausgaben und Zustandsänderungen auf Verhaltenskorrektheit zu überprüfen. Dieser Prozess erleichtert auch die Validierung des Systemverhaltens im Hinblick auf beabsichtigtes Verhalten. Komplexe Systeme wie reaktive Systeme haben viele Verhaltensaspekte, darunter Funktionalität, Timing, Sicherheit und Leistung. Der Testprozess sollte die Ergebnisse gegen Anforderungen für all diese Verhaltensweisen überprüfen. Wie bereits in [ZH02] erläutert wird, ist die Überprüfung solcher Dokumente sehr zeitaufwendig. Daher ist es notwendig, diese Informationen vorher zu vereinfachen und überschaubar zu gestalten.

2.3 Die Interpretation der Ergebnisse

Mögliche Testergebnisse

(Übersetzt aus [FB97]) Ein Test weist ein negatives Ergebnis auf, wenn während seiner Ausführung ein Fehler auftritt, was bedeutet, dass das Programm abstürzt oder eine definierte Eigenschaft verletzt wird. Ein Test hat ein positives Ergebnis,

wenn eine Reihe von Tests keine Fehler ergibt und diese Reihe von Tests gemäß einer definierten Abdeckungsmetrik als „vollständig“ gilt. Ein Test wird als „unvollständig“ betrachtet, wenn eine Reihe von Tests keine Fehler aufzeigt, jedoch die Abdeckungsmetrik nicht vollständig erfüllt ist.

Shrinking

Wie bereits in Kapitel 2 mehrfach erwähnt, ist es erforderlich, dass das Testverfahren solche Fehler eigenständig aufdecken kann. Dennoch obliegt es dem Tester, die Ursache dieser Fehler zu ermitteln und anschließend eine vereinfachte Variante zur Reproduktion des Fehlers zu konstruieren. In der Untersuchung von [Gol+24] wird dieser Prozess als „Shrinking“ bezeichnet, wobei festgestellt wurde, dass dieser Shrinking-Prozess mit Schwierigkeiten verbunden ist. Häufig sind die dabei erzeugten Bug-Berichte zur Reproduktion bestimmter Fehler viel zu umfangreich, um sie angemessen analysieren zu können, und müssen daher zunächst auf eine für den Menschen verständlichere Variante reduziert werden. In [ZH02] wird dieser Konflikt wie folgt beschrieben:

> „A bug report should be as specific as possible, such that the engineer can recreate the context in which the program failed. On the other hand, a test case should be as simple as possible because a minimal test case implies a most general context.“

Validierung von PBT-Tests

Bei PBT muss der Entwickler nicht nur feststellen können, ob ein Test fehlschlägt, sondern auch beurteilen, ob das Bestehen eines Tests tatsächlich darauf hinweist, dass die Software korrekt funktioniert [Gol+24]. In Kapitel 2.1 wird hervorgehoben, dass Random Testing nicht darauf abzielt, ein Programm vollständig zu testen. Daher besteht die Möglichkeit, dass bestimmte Eingabeparameter oder Sequenzen, die einen Fehler verursachen würden, nicht erzeugt werden. Es ist ebenfalls möglich, dass ein Generator aufgrund mangelhafter Programmierung nicht in der Lage ist, die erforderlichen Eingabeparameter zu erzeugen [Gol+24]. Das Paper von [Gol+24] beschreibt darüber hinaus verschiedene Techniken zur Validierung von PBT-Tests, die aus der Befragung hervorgingen:

Mutation Testing

> „Some participants intentionally added bugs to their code and checked that their tests successfully found those bugs. This technique is standard in the testing literature [Pap+19; PI18] and used in testing benchmarks such as Magma [HHP20] and Etna [Shi+23].“

Example Inspection

> „An even simpler way to assess a generator’s distribution is to look at a handful of examples that the generator produces; one participant (P26) even

designed a small utility to graphically render one example at a time, to make them easier to understand at a glance.“

Code Coverage

„Participants evaluated the code coverage achieved by their tests and used code coverage measurements as an indication that their properties were thoroughly exploring the space of program behaviors.“

Property Coverage

„Another participant measured coverage not of the system under test, but of the property itself. This is a weaker measurement, since it does not say anything about the system under test, but it is much easier to make because it can be measured without complex tooling.“

In [Gol+24] wird darauf hingewiesen, dass einige Teilnehmer ergänzend zum PBT-Verfahren auch Unit-Tests einsetzen, um potenzielle Lücken der beiden Testmethoden gegenseitig zu kompensieren. In [Gol+24] wird zudem betont, dass es grundsätzlich keine gute Idee sei, ein einziges Test-Tool isoliert zu verwenden, wobei insbesondere auf QuickCheck Bezug genommen wird.

3 LLMs und deren Verwendung bei Softwaretests

In diesem Kapitel wird zunächst die Netzwerkarchitektur, die von vielen heutigen Modellen genutzt oder adaptiert wird, untersucht, um ein grundlegendes Verständnis für die Funktionsweise großer Sprachmodelle zu vermitteln. Anschließend werden grundlegende Prinzipien zur Erstellung von Prompts, auch bekannt als Prompt-Engineering, erläutert. Diese Strategien werden auch bei der Implementierung des Prototyps angewendet. Abschließend wird noch die State-of-the-Art-Applikation Fuzz4All betrachtet.

3.1 Architektur von LLMs

Im Jahr 2017 wurde mit dem Paper „Attention is all you need“ [Vas+17] eine neue Netzwerkarchitektur vorgestellt. Dieser Transformer basiert ausschließlich auf dem Attention-Mechanismus und verzichtet vollständig auf Rekursion und Faltung. Diese Architektur übertrifft die Leistung der damals bekannten neuronalen Netzwerke, wie beispielsweise Recurrent Neural Networks (RNN) [MJ+01] sowie auf RNN basierende Architekturen wie Gated Recurrent Units (GRU) [DS17] und Long Short-Term Memory (LSTM) [HS97]. Im Paper [Vas+17] wird dazu ausgeführt: „Experiments [...] show these models to be superior in quality while being more parallelizable and requiring significantly less time to train.“ Während frühere Modelle auf eine bestimmte Kontextlänge beschränkt waren, ermöglicht die Einführung der Transformer-Architektur, und der damit verbundenen Attention Mechanismen, die Nutzung einer deutlich größeren Kontextlänge. So wird bereits an Modellen gearbeitet, die eine Kontextlänge von bis zu 1 Milliarde Tokens besitzen sollen [Din+23]. Um das Funktionsprinzip und die Arbeitsweise von Modellen, die auf dieser oder einer ähnlichen Architektur basieren, zu verstehen, werden in diesem Kapitel die erforderlichen theoretischen Grundlagen und die verschiedenen Prozesse in chronologischer Reihenfolge behandelt.

Begriffsdefinitionen

Bevor diese Transformer-Architektur vorgestellt wird, sollen zunächst die Begriffe Checkpoint, Architektur und Modell definiert werden. Die hier verwendeten Definitionen finden auch im Natural Language Processing (NLP) Kurs von Huggingface [Hug22a] Anwendung. Checkpoints beziehen sich auf die Gewichte, die von einer Architektur geladen werden. Die Architektur stellt das strukturelle Gerüst des Modells dar und legt die Spezifikationen für jede Schicht sowie alle Operationen fest, die während des Prozesses durchlaufen werden. Der Begriff Modell ist

weniger präzise als Architektur oder Checkpoint und kann beide Bedeutungen umfassen.

Encoder

Tokenizer

Im Folgenden werden die verschiedenen Prozesse, wie sie in Abbildung 2 dargestellt werden, behandelt. Damit der Encoder den Eingabetext weiterverarbeiten kann, muss dieser in ein entsprechendes Format transformiert werden. Wie in [WK92] ausgeführt wird, besteht der erste Schritt beim NLP darin, dass der an das Modell übergebene Text zerlegt wird. Diese Zerlegung kann beispielsweise durch einen Word-Based Tokenizer [Mik+13; LY18] erfolgen, der den Text in einzelne Wörter unterteilt, oder durch einen Character-Based Tokenizer [Tay+21],

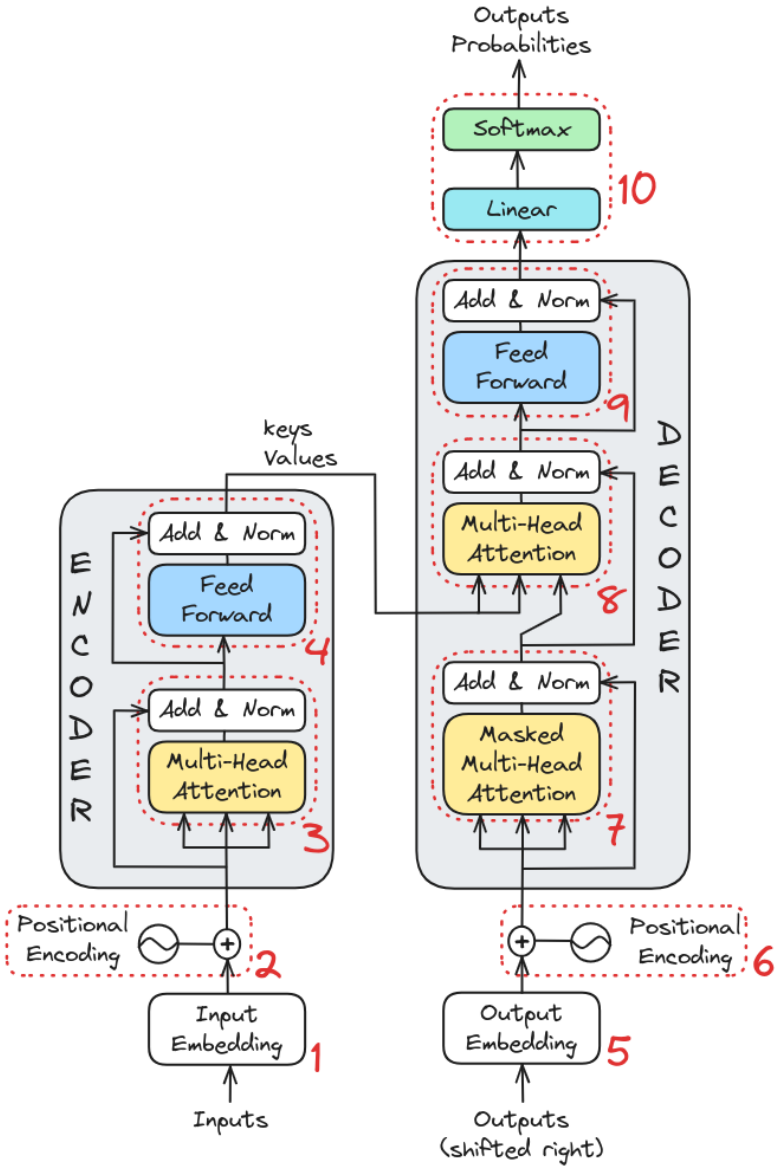


Abbildung 2: Transformer - Model Architektur [Vas+17]

der den Text in einzelne Zeichen aufspaltet. Angenommen, ein Word-Based Tokenizer wird verwendet, so wird eine Liste erstellt, in der jeder Eintrag ein Token in Form eines Wortes enthält. Wie in [Pat+23] ausgeführt wird, besteht der nächste Schritt darin, jedem Token aus dieser Liste auf einen nummerischen Vektor abzubildet. Diese Vektoren müssen stets die gleiche Größe aufweisen, wobei der Dimensionenbereich üblicherweise zwischen 100 und 500 liegt. Kurz zusammengefasst wird dadurch die Bedeutung des Wortes im Text dargestellt. In [Pat+23] wird hierzu Folgendes geschrieben:

Input Embedding (1)

> „In embeddings derived from such continuous vector space, the focus is more on computing the semantics of individual words by looking at the context in which they appear. This technique considers the effect that neighboring words can have on the given word and how these relationships affect the meaning of a word. These new word vectors are context-sensitive, can identify synonyms and antonyms, and construct analogies and categories of words, which was impossible in earlier approaches. Word vectors capture words' meanings (literal and implied) and represent them using dense floating-point values. They represent the semantics as well as the syntactic aspects of the word."

Positional Encoding (2)

Da das Modell weder Rekursion noch Faltung verwendet, müssen Informationen über die relative oder absolute Position der Tokens in der Sequenz hinzugefügt werden, damit das Modell die Reihenfolge der Sequenz auch nutzen kann [Vas+17], weil die Bedeutung eines Wortes oft stark von seiner Position im Satz abhängt. Dies ist erforderlich, um sicherzustellen, dass das Modell Wörter, die nahe beieinander liegen, als „nahe" und Wörter, die weit entfernt sind, als „weit entfernt" behandelt [Geh+17]. Zu diesem Zweck werden sogenannte Positional Embeddings erzeugt. Diese Embeddings müssen die gleiche Dimension wie die Input Embeddings haben, damit sie addiert werden können [Vas+17]. Das Ergebnis dieser Addition kann dann als Input für den Encoder bzw. Decoder verwendet werden. Für die Berechnung der Positional Embeddings werden oft die trigonometrischen Funktionen Kosinus und Sinus verwendet [WC20]. Die Verwendung von Positionskodierungen, insbesondere durch Sinus- und Kosinusfunktionen, ermöglicht es dem Modell, relative Positionen innerhalb der Sequenz zu erfassen. Diese Funktionen gewährleisten konsistente Frequenzen und Phasenverschiebungen über verschiedene Positionen hinweg, wodurch das Modell in der Lage ist, die relativen Positionen der Tokens innerhalb der Sequenz effizient zu lernen [Kaz19].

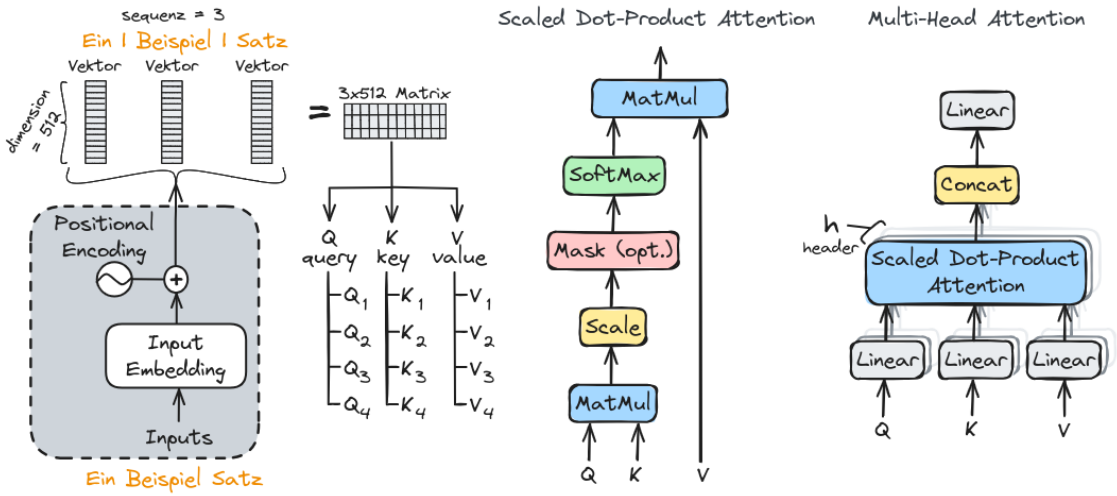


Abbildung 3: Transformer - Attention-Mechanismus [Vas+17]

Self Attention (3)

Der Encoder setzt sich aus zwei Hauptkomponenten zusammen. Zunächst werden das Multi-Head Attention Modul und die damit verbundene Scaled Dot-Product Attention [Vas+17] betrachtet, welche im Folgenden als Self-Attention Funktion beschrieben wird. Durch die Anwendung der Self-Attention Funktion kann das Modell die Beziehungen zwischen den Wörtern erfassen und abbilden [GLT21]. Wie in Abbildung 3 dargestellt, beginnt der Prozess mit einer Matrixmultiplikation von Q und der transponierten Matrix K. Hierbei handelt es sich bei Q (Query), K (Key) und V (Value) um Kopien der zuvor erzeugten Input-Matrix. Die daraus resultierende Matrix wird auch als Score-Matrix bezeichnet und enthält Informationen darüber, wie stark jedes Wort mit jedem anderen Wort in Beziehung steht. Nach der Matrixmultiplikation wird das Ergebnis herunterskaliert, indem es durch die Wurzel der Dimension der Spalten (Anzahl der Spalten in der Schlüsselmatrix) geteilt wird. Nachdem das skalierte Ergebnis vorliegt, wird eine Softmax-Funktion [Ban+20] angewendet. Diese Funktion wird typischerweise am Ende eines neuronalen Netzwerks verwendet, um die Ausgaben in einer Wahrscheinlichkeitsverteilung zu transformieren. Mittlerweile wird die Softmax-Funktion aber auch in anderen Operationen, wie beim Attention-Mechanismus, eingesetzt. Die Softmax-Funktion ist definiert durch die folgende Formel:

$$softmax\left(z\right)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

In diesem Kontext ist z ein Vektor mit k reellen Zahlen $(z_1, z_2, \ldots, z_k)$. Für jedes Element z_i im Vektor z wird die Exponentialfunktion e^{z_i} berechnet. Anschließend werden diese Exponentialwerte normalisiert, indem sie durch die Summe aller Exponentialwerte dividiert werden. Dadurch entsteht eine Matrix mit Attention-Weights bzw. Wahrscheinlichkeitswerten, die im Bereich von 0 bis 1 liegen. Die

Softmax-Funktion verstärkt hierbei hohe Werte weiter und dämpft niedrige Werte, wodurch das Modell besser bestimmen kann, welche Wörter stärker berücksichtigt werden müssen. Abschließend wird die Matrix mit den Gewichten noch mit V Multipliziert. Dieses Ergebnis enthält schließlich Informationen zu den Beziehungen zwischen den Wörtern. Dieser Vorgang wird in [Vas+17] auch mit der folgenden Formel beschrieben:

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

Multi-Head Attention

Im folgenden Schritt wird nun das Verfahren der Multi-Head Attention beschrieben, welches das zuvor erläuterte Verfahren der Self-Attention nutzt. Hier werden die Matrizen Q, K und V vorher noch in h Teile geteilt. Nehmen wir das Beispiel aus Abbildung 3 mit der 3x512-Matrix und legen fest, dass $h = 4$ ist. Wir erhalten somit Matrizen der Dimension 3x128, welche im weiteren Verlauf mit Q_i, K_i und V_i bezeichnet werden. Diese kleineren Matrizen werden genutzt, um mit der Self-Attention Funktion die Header mit den Wahrscheinlichkeitswerten zu berechnen:

$$head_i = Attention\,(Q_i, K_i, V_i)$$

Diese Header werden anschließend verkettet und erneut projiziert, was zu den endgültigen Werten führt. Der daraus gewonnene Vorteil lässt sich wie folgt beschreiben: „Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions“ [Vas+17].

Add & Norm

In Abbildung 2 sind auch mehrere Layer mit der Beschriftung „Add & Norm“ dargestellt. Der Add-Schritt stellt eine Residualverbindung [Bal+17] dar, die sicherstellt, dass ein stärkeres Informationssignal durch das Netzwerk geleitet wird. Der Normalisierungsschritt sorgt dafür, dass die Werte wieder auf einen Bereich zwischen 0 und 1 gebracht werden, was ein stabileres Training ermöglicht. Aufgrund von Backpropagation können die Gradienten im Laufe der Zeit immer kleiner werden und schließlich den Wert 0 erreichen, was dazu führen würde, dass das Modell nicht mehr weiter lernt. Das „Add & Norm“-Layer verhindert dies, indem er die Stabilität der Gradienten aufrechterhält.

Feed-Forward-Network (4)

Das Feed-Forward-Network (FFN) besteht aus zwei linearen Transformationen, zwischen denen eine Rectified Linear Unit (ReLU)-Aktivierungsfunktion liegt. In [HSS15] wird folgendes dazu geschrieben:

> „Deep learning employs the gradient algorithm, however this traps the learning into the saddle point or local minima. To avoid this difficulty, a rectified linear unit (ReLU) is proposed to speed up the learning

> convergence. However, the reasons the convergence is speeded up are not well understood."

Während die linearen Transformationen für verschiedene Positionen identisch bleiben, werden in den einzelnen Layern unterschiedliche Parameter verwendet [Vas+17]. Basierend auf einer Diskussion auf StackExchange [htt] wird die Notwendigkeit des FFN-Layers noch folgendermaßen begründet: Multi-Head-Attention kann als eine Funktion für die Berechnung von Durchschnittswerten betrachtet werden. Wenn es kein FFN gäbe, würde dieser Prozess in den verschiedenen Layern stets gleich oder ähnlich funktionieren. Das Hinzufügen des FFNs bewirkt, dass sich jedes von ihnen wie ein separates kleines Modell verhält und somit für eine gewisse Variabilität sorgt.

Decoder

(5, 6)

Beim Decoder werden sogenannte Output Embeddings erzeugt und mittels Positional Encoding weiterverarbeitet. Dieser Vorgang erfolgt analog zu den zuvor beschriebenen Input Embeddings des Encoders. Das dabei erzeugte Ergebnis wird diesmal an ein Masked Multi-Head Attention Layer übergeben, wodurch ein kausales Modell entsteht. Das heißt, die Ausgabe an einer bestimmten Stelle kann sich nur auf Wörter beziehen, die vor ihm auftauchen und nicht danach. Dies ist möglich, indem man, wie in Abbildung 3 dargetellt, bei der Self-Attention Funktion die zuvor erzeugte Matrix $Q * K^T$ zusätzlich noch mit einer Look-Ahead Mask Matrix addiert, die alle Werte rechts von der Diagonalen auf $-\infty$ setzt. Alle $-\infty$ Werte werden somit nach Anwendung der Softmax-Funktion schließlich auf 0 gesetzt.

Masked Multi-Head Attention (7)

(8, 9)

Das Ergebnis des Encoders wird anschließend mit dem Ergebnis der Mask Multi-Head Attention Funktion des Decoders an die Multi-Head Attention Funktion des Decoders übergeben. Auch hier ist die Vorgehensweise beim Multi-Head Attention Layer und dem FFN-Layer die selbe wie beim Encoder. Die Ausgabe des Decoders wird für gewöhnlich auch Logits genannt und wird abschließend noch durch das Linear Layer und dem Softmax Layer weiterverarbeitet. Dabei ist das Linear Layer dafür zuständig, die Logits auf unser Vokabular zu projiziert und nach Anwendung der Softmax Funktion kann dann das Wort mit der höchsten Wahrscheinlichkeit gewählt werden. Das erzeugte Wort kann dann, wie in Abbildung 2 dargestellt, rechts an den Output angehangen werden, damit das nächste Wort vorhergesagt werden kann. Der Dekodierungsprozess wird fortgesetzt, bis entweder ein spezieller Token, der das Ende des Textes signalisiert, generiert wird, oder die maximale Anzahl an Tokens erreicht ist.

Linear Layer & Softmax Layer (10)

3.2 Optimierung der Ausgabe ohne Training

Wie in Kapitel 3.1 erläutert, kodieren LLMs den Eingabetext in einen hochdimensionalen Vektorraum, wobei semantische Beziehungen zwischen Wörtern und Sätzen erhalten bleiben. Der Decoder nutzt diese Darstellung, um nach und nach Tokens und somit eine Antwort zu generieren, die sich an den erlernten statistischen Mustern orientiert [Ben+21]. Die Qualität der Antwort kann dabei durch verschiedene Faktoren beeinflusst werden, wie z. B. durch den bereitgestellten Prompt, den verwendeten Hyperparametern des Modells und von der Vielfalt der Trainingsdaten [Che+23]. In diesem Kapitel werden sowohl die grundlegenden Prinzipien des Prompt-Engineerings erläutert als auch die Auswirkung von einzelnen Modellparametern beschrieben.

Prompt Engineering

[Che+23] Die Praxis des Prompt-Engineerings, die ursprünglich als grundlegende Methode begann, hat sich mittlerweile zu einem strukturierten Forschungsbereich entwickelt, der mit eigenen Methoden und bewährten Verfahren ausgestattet ist. Prompt-Engineering bezeichnet die systematische Gestaltung und Optimierung von Eingabeaufforderungen, auch bekannt als Prompts, mit dem Ziel, die Antworten von LLMs gezielt zu lenken und die Genauigkeit, Relevanz und Kohärenz der generierten Ergebnisse zu gewährleisten. Die Konstruktion von Prompts kann nach Studien von [Kad+23], [Lu+21] und [WP21] signifikante Unterschiede in der Ausgabe bewirken. Ein gut konstruierter Prompt kann sogar Herausforderungen wie Maschinenhalluzinationen entgegenwirken, wie in Studien von [May+20] und [Bub+23] hervorgehoben wird. Die Tabelle 1 listet einige dieser Techniken auf. Zuerst werden grundlegende Techniken wie Zero-, One- und Few-Shot in diesem Kapitel vorgestellt, gefolgt von einer fortgeschrittenen Technik namens Chain-of-Thought (CoT).

Einfache Techniken
Zero-Shot
One-Shot
Few-Shot
Role Prompting

(a) Einfache Techniken

Fortgeschrittene Techniken
Chain of thought
- Zero-shot chain of thought
- Golden chain of thought
Self-consistency
Generated knowledge
Least-to-most prompting
Tree of thoughts
Graph of thoughts
Retrieval augmentation
Use plugins to polish the prompts

(b) Fortgeschrittene Techniken

Tabelle 1: Prompt-Engineering Techniken nach [Che+23]

Allgemeingültige Prinzipien

Bevor man die oben genannten Techniken vorstellt, werden zunächst allgemeine Prinzipien behandelt, die für alle Ansätze gleichermaßen wirksam sind. Untersuchungen wie die von [Luo+19] und [YJ18], die mit GPT-4 durchgeführt wurden, haben gezeigt, dass das Sprachmodell dazu neigt, übermäßig allgemeine Antworten zu generieren, insbesondere wenn die Eingabeaufforderung nicht ausreichend detailliert ist. Daher ist eine umfassende Beschreibung unabdingbar, um präzisere und relevantere Ergebnisse zu erzielen [Yan+19]. Darüber hinaus betont [Che+23], dass die Beschreibungen nicht nur umfassend, sondern auch so „klar und präzise“ wie möglich sein sollten. Dieser Aspekt wird durch die für das Training verwendeten Daten begründet. Die meisten LLMs werden mit einer breiten Palette von Textdaten trainiert, die von verschiedenen Autoren stammen. Wenn die Eingabeaufforderungen breit gefasst oder ungenau sind, tendiert die Ausgabe dazu, ebenfalls allgemein zu sein. Dies kann zwar in verschiedenen Kontexten anwendbar sein, ist jedoch möglicherweise nicht optimal für eine spezifische Anwendung.

One- & Few-Shot

Zunächst werden die beiden gängigen Techniken One-Shot und Few-Shot betrachtet. Bei der One-Shot-Technik enthält der Prompt genau ein Beispiel mit einer entsprechenden Antwort, die zuvor von einem Menschen formuliert wurde [Che+23]. Dieses Beispiel dient hierbei als Vorlage, mit dem Ziel die Antwort des Modells zu lenken. Enthält ein Prompt mehr als ein Beispiel, so spricht man von Few-Shot [Log+21]. Nach [Che+23] hängt die Wahl zwischen One-Shot- und Few-Shot-Prompting häufig von der Komplexität der Aufgabe und den Fähigkeiten des Modells ab.

Zero-Shot

Zero-Shot Anweisungen sind hingegen Prompts, die keine Beispiele enthalten. Wie in [RM21] festgestellt wurde, ist es sogar möglich, durch sorgfältig ausgearbeitete Zero-Shot Anweisungen bessere Ergebnisse zu erzielen als mit Few-Shot Anweisungen. Damit werden auch Erkenntnisse aus der Arbeit von [Bro+20] in Frage gestellt, in denen behauptet wird, dass Sprachmodelle am besten mit Few-Shot Techniken lernen. Möglicherweise hängt dies jedoch auch von der spezifischen Kombination aus Modell und der zu bewältigenden Aufgabe ab. In Abbildung 4 wird der Unterschied zwischen einer Zero-Shot und einer One-Shot Anweisung nochmals veranschaulicht.

Role-Prompting

Neben der Verwendung von Beispielen in Prompts kann das Modell auch durch die Zuweisung einer bestimmten Rolle geleitet werden, um die generierten Antworten zu beeinflussen [Che+23]. Beispielsweise kann dem Modell die Rolle eines „Python-Code-Generators“ zugewiesen werden, um gezielt Code in der Programmiersprache Python zu erzeugen. Durch die Zuweisung einer spezifischen Rolle an das Modell erhöht sich die Wahrscheinlichkeit, dass kontextgenauere Antworten generiert werden, wenn eine Frage oder Aufgabe gestellt wird

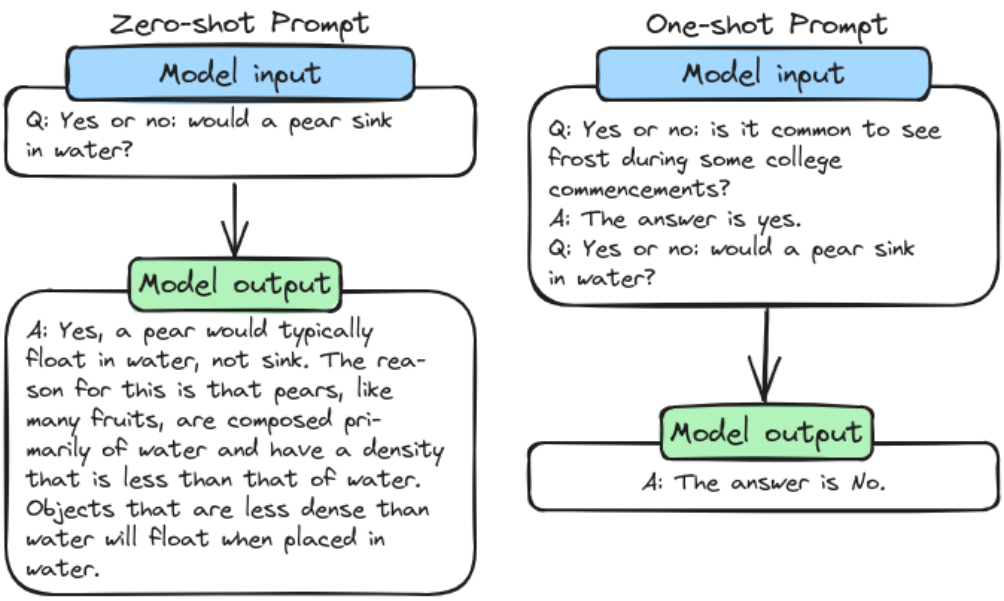


Abbildung 4: Zero- vs. One-Shot [Che+23]

```
[
  {"role": "user", "content": "Hello, how are you?"},
  {"role": "assistant", "content": "I'm fine. How can I help you today?"},
  {"role": "user", "content": "I'd like to show off how chat templating works!"}
]
```

Listing 2: Chat-Template Beispiel nach [Hug22b]

[Van23]. In [Zha+23] wird diese Technik demonstriert, indem das Modell als Assistent oder sachkundiger Experte fungiert. Eine Besonderheit der Transformer-Bibliothek von Huggingface ist die Bereitstellung spezifischer `«chat_template»`-Funktionen. Diese Funktionen ermöglichen es, Prompts strukturiert und systematisch zu gestalten, wie in Listing 2 veranschaulicht.

Chain of Thought

Der Begriff CoT-Prompting bezieht sich auf die Bereitstellung von Zwischenschritten, die eine bestimmte Form des logischen Denkens widerspiegeln [Che+23]. Insbesondere bei Aufgaben zur logischen Schlussfolgerung hat sich diese Methode bewährt und zu einer Verbesserung der Modellausgaben geführt [Wu+23; Zha+22; Lew+22]. Auch hier gibt es wieder verschiedene Herangehensweisen. Eine der einfachsten besteht darin, Anweisungen wie „Denken wir Schritt für Schritt“ direkt in den Prompt einzubinden, wie es zum Beispiel in Arbeiten von [Zho+22] oder [Koj+22] demonstriert wird. Alternativ ist es auch möglich, eine Reihe von manuellen Demonstrationen vorzugeben, die jeweils aus einer Frage und einer Argumentationskette bestehen, die zu einer Antwort führt [Zho+22; Lee+23]. Gemäß [Che+23] erhält der „Denkprozess“ des Modells eine deutliche

Struktur, was es den Benutzern erleichtert, die Antworten zu verstehen und auch besser nachvollziehen zu können, wie das Modell zu diesen Schlussfolgerungen gelangt.

Resampling

Resampling bezeichnet die wiederholte Ausführung der Inferenz mit demselben Modell und demselben Prompt, mit dem Ziel, eine Ausgabe zu erzeugen, die sich von der vorherigen unterscheidet [Che+23]. Dieser Vorgang kann dabei beliebig oft wiederholt werden, wobei am Ende das beste Ergebnis ausgewählt wird. In [Che+23] wird dazu geschrieben, dass sich dies aufgrund des nichtdeterministischen Charakters von LLMs empfiehlt. Allerdings hängt der deterministische Charakter auch von den verwendeten Modellparametern ab. Werden Parameter eingesetzt, die das Modell nahezu deterministisch machen, ist Resampling nicht empfehlenswert, da derselbe Prompt stets zur selben Antwort führen würde. Unter Verwendung geeigneter Modellparameter kann dieser Ansatz jedoch dazu beitragen, die inhärente Variabilität in den Antworten des Modells zu überwinden und die Wahrscheinlichkeit einer qualitativ hochwertigen Ausgabe zu erhöhen [Hol+19].

Dreifache Anführungszeichen

Eine weitere in der Arbeit von [Che+23] erwähnte Technik ist die Verwendung von dreifachen Anführungszeichen, um mehrere separate Teile innerhalb eines Prompts zu trennen oder zu umschließen. Diese Methode hat sich speziell beim ChatGPT-Modell von OpenAI als effektiv erwiesen. Die Wirksamkeit dieser Technik muss daher bei anderen Modellen erst noch untersucht werden.

Modellparameter

Ein weiterer wesentlicher Faktor bilden die zuvor genannten Modellparameter. In [Che+23] werden insbesondere die Parameter *temperature*, *top-k* und *top-p* behandelt. Zunächst wird der *temperature*-Parameter betrachtet, der die von der Softmax-Funktion erzeugten Werte beeinflusst [Hol+19]. Wie in Kapitel 3.1 erklärt, liefert die Softmax-Funktion Wahrscheinlichkeitswerte, die angeben, mit welcher Wahrscheinlichkeit ein bestimmter Token als nächstes ausgewählt wird. Ein höherer *temperature*-Wert führt hierbei zu zufälligeren Ausgaben, während ein niedrigerer Wert die Ausgabe deterministischer macht [AHS85; FG17]. Dies wird auch noch einmal in Abbildung 5 veranschaulicht. In den meisten Fällen wählt das LLM unter Berücksichtigung der Einzelwahrscheinlichkeiten zufällig aus einer Liste möglicher Tokens. Bevor die Parameter *top-k* und *top-p* sowie weitere erläutert werden, muss zunächst geklärt werden, was unter einer Dekodierstrategie verstanden wird.

Dekodierstrategien

Als Dekodierstrategie bezeichnet man verschiedene Methoden, mit denen der Decoder des Transformers den nächsten Token bzw. seine Antwort generiert. Hierbei gibt es eine Vielzahl an Möglichkeiten. In Tabelle 2 werden alle mög-

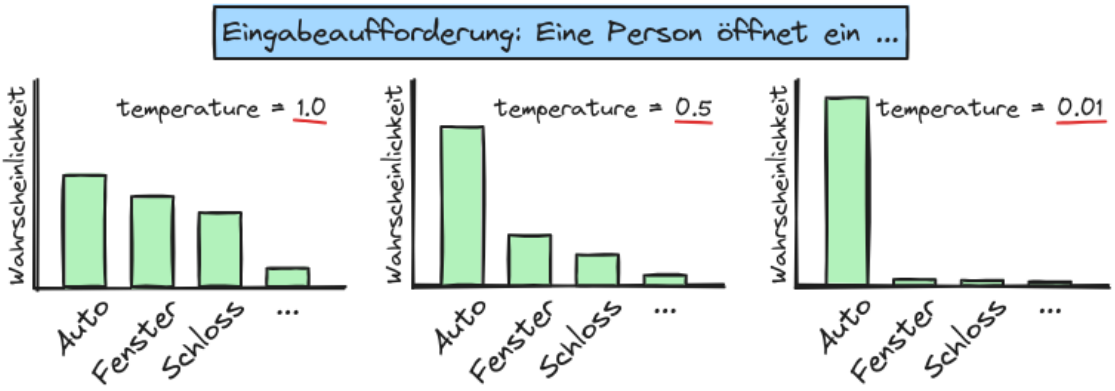


Abbildung 5: Einfluss des *temperature*-Parameters auf die durch die Softmax-Funktion berechneten Einzelwahrscheinlichkeiten

Dekodierstrategien	**Hauptparameter, die das Verhalten ermöglichen und steuern**
Greedy Search	num_beams = 1 do_sample = False
Contrastive search	penalty_alpha>0 top_k>1
Multinomial sampling	do_sample=True num_beams=1
Beam-search decoding	do_sample=False num_beams>1
Beam-search multinomial sampling	do_sample=True num_beams>1
Diverse beam search decoding	diversity_penalty>0 num_beams>1 num_beam_groups>1

Tabelle 2: Dekodierstrategien und deren Hauptparametern, die das Verhalten ermöglichen und Steuern

lichen Strategien aufgeführt, die durch Anpassungen der Modellparameter verwendet werden können. Dabei werden auch die Hauptparameter als auch die entsprechenden Bedingungen mit aufgeführt, die für die verschiedenen Strategien erforderlich sind und mit denen das entsprechende Verhalten gesteuert werden kann. Neben diesen aufgelisteten Verfahren werden von der Transformer-API auch noch die beiden Verfahren Constrained Beam-Search [Hu+15] und Speculative Decoding [LKM22] unterstützt, die in dieser Arbeit allerdings nicht weiter betrachtet werden. Darüber hinaus wird in diesem Abschnitt nicht jeder einzelne Parameter im Detail behandelt. Für eine detailliertere Betrachtung verweise ich auf einen Auszug aus der bereitgestellten Konfigurationsdatei (siehe Anlage E: Modellparameter), in der die einzelnen Parameter kurz erläutert werden. Es sollte jedoch beachtet werden, dass nicht jede Architektur jede Dekodierstrategie

unterstützt. Als Beispiel soll hierfür das StarCoder-Modell [Li+23] genannt werden, welches die Architektur „GPTBigCodeForCausalLM“ [All+23] verwendet. Mit dieser Architektur kann die Strategie Contrastive Search nicht verwendet werden.

Greedy Search

Mit der Greedy Search Strategie werden vom Decoder stets die Token ausgewählt, welche die höchste Wahrscheinlichkeit besitzen. Dieser Ansatz zählt zu den einfachsten Methoden, jedoch tendieren die generierten Ausgaben häufig dazu, zusammenhangslos, redundant oder widersprüchlich zu sein [Sha+17].

Beam Search

Während der Decoder bei Greedy Search bei jedem Schritt stets den erst besten Token wählt und somit nur eine Sequenz als Lösung generiert, werden mit Beam Search mehrere verschiedene Sequenzen zuerst erzeugt und betrachtet. Am Ende wird schließlich die Sequenz mit der höchsten Gesamtwahrscheinlichkeit gewählt.

Multinomial Sampling

Sampling bedeutet grundsätzlich, dass der Decoder bei jedem Schritt den nächsten Token zufällig wählt, wobei die Auswahl auch von den bedingten Wahrscheinlichkeiten der Tokens abhängt. Die resultierenden Texte sind dabei deutlich vielfältiger, jedoch wirken sie oft zusammenhangslos. Um diesem Effekt entgegenzuwirken, kann die Wahrscheinlichkeitsverteilung der Tokens durch Anpassung des *temperature*-Werts der Softmax-Funktion klarer gemacht werden. Eine weitere Möglichkeit besteht darin, dass der Decoder aus einer festen Menge der wahrscheinlichsten Tokens wählt. Das Verfahren wird auch als *top-k* Sampling [FLD18] bezeichnet. Ein ähnlicher Ansatz wird mit Nucleus Sampling [Hol+19], auch bekannt als *top-p* Sampling, verfolgt. Hierbei wird keine feste Menge definiert, sondern eine Wahrscheinlichkeit p. Bei diesem Verfahren wählt der Decoder aus der kleinstmöglichen Menge von Tokens, deren kumulative Wahrscheinlichkeit die Wahrscheinlichkeit p überschreitet.

Beam Search Multinomial Sampling

Beam Search Multinomial Sampling [Sha+17] kombiniert die beiden Dekodierstrategien Beam Search und Multinomial Sampling. Dabei werden, ähnlich wie bei der Beam Search-Strategie, mehrere Sequenzen generiert. Jedoch wendet der Decoder bei der Generierung der Sequenzen zufallsbasierte Methoden an.

Diverse Beam Search

Mit der Diverse Beam Search Strategie [Vij+16] werden mehrere Gruppen erzeugt, die entsprechend auch mehrere Sequenzen beinhalten. Ziel bei diesem Verfahren ist eine größere Diversität zwischen den Gruppen zu erzwingen. Die Sequenzen innerhalb einer Gruppe können dabei wieder eine größere Ähnlichkeit aufweisen.

Contrastive Search

Die Schlüsselideen der Contrastive Search Strategie werden im Paper [Su+22] wie Folgt beschrieben:

> „At each decoding step, the key ideas of contrastive search are (i) the generated output should be selected from the set of most probable candidates predicted by the model; and (ii) the generated output should be discriminative enough with respect to the previous context. In this way, the generated text can (i) better maintain the semantic coherence with respect to the prefix while (ii) avoiding model degeneration.“

Bewertung der Wirksamkeit von Methoden

Zum Abschluss dieses Unterkapitels sollen einige Verfahren vorgestellt werden, mit denen die Wirksamkeit der verwendeten Methoden beurteilt werden kann. [Che+23] stellt hierbei allgemein subjektive und objektive Bewertungsmethoden vor. Beim subjektiven Verfahren werden die generierten Inhalte von Menschen überprüft. Gemäß [Hol+19] umfasst die Qualität der Modellausgaben typischerweise Aspekte wie Flüssigkeit, Genauigkeit, Neuartigkeit und Relevanz. Je nach spezifiziertem Ziel können jedoch auch individuelle Kriterien festgelegt werden. [Che+23] betont jedoch, dass derartige Bewertungsmethoden per Definition subjektiv sind und anfällig für Inkonsistenzen sein können. Zudem gelten sie als kostspieliger und zeitaufwändiger. Trotz dieser Einschränkungen werden subjektive Verfahren für die Bewertung generierter Ausgaben als zuverlässiger eingeschätzt. Bei der objektiven Evaluierung, auch bekannt als automatische Evaluierungsmethoden, werden Algorithmen des maschinellen Lernens eingesetzt, um die Qualität der von LLMs generierten Texte automatisiert zu bewerten [Che+23]. Wie man in [SMK22] feststellt, erfassen diese automatisierten Metriken oft nicht vollständig die Bewertungsergebnisse menschlicher Bewerter und sollten daher mit Vorsicht interpretiert werden. Trotzdem ist eine objektive Bewertung im Vergleich zu subjektiven Verfahren weniger kostspielig und zeitsparender.

Bei der Erarbeitung einer geeigneten Lösung durch die Verwendung verschiedener Prompts ist es ratsam, diese zu versionieren. Zu diesem Zweck stehen bereits fertige Softwarelösungen und Tools zur Verfügung, wie beispielsweise das Open-Source-Projekt Pezzo [Wei23]. Die während der Evaluation mit verschiedenen Prompts oder Techniken erzielten Ergebnisse können anschließend miteinander verglichen werden. Ein dafür geeignetes Verfahren ist z. B. das A/B-Testing [Qui+24]. Es handelt sich hierbei um einen statistischen Ansatz, der verwendet wird um zwei oder mehr Tests mit unterschiedlichen Setups zu vergleichen, um festzustellen, welche Version effektiver ist.

3.3 Benchmarks zur Messung der Leistung von Modellen

Um die Leistung eines Modells zu bewerten und auch mit anderen Modellen vergleichen zu können, werden für gewöhnlich Benchmarks zur Evaluierung verwendet. Dabei wird dem Modell eine Eingabe übergeben und man überprüft ob die dabei erzeugte Ausgabe mit der korrekten Antwort übereinstimmt. Bei solchen Benchmarks werden verschiedene Fähigkeiten des betrachteten Modells gemessen und am Ende in Form von Zahlen dargestellt. Einige bekannte Benchmarks sind z. B. ARC [MOM23], HellaSwag [Zel+19], MMLU [Hen+20], TruthfulQA [LHE21], Winogrande [Sak+19] und GSM8K [Cob+21].

abgleichsbasierte Metriken

Frühere Ansätze zur Bewertung von Code-LLMs nutzten abgleichsbasierte Metriken, bei denen die Modellausgaben mit einer Referenzlösung verglichen wurden [Che+21]. Es wurde jedoch festgestellt, dass sich dieses Verfahren für Programmcode nicht besonders eignet [Ren+20]. Infolgedessen haben sich Arbeiten wie [Lac+20] und [Kul+19] auf die Überprüfung der funktionalen Korrektheit konzentriert, bei der ein Codebeispiel als korrekt gilt, wenn es eine Reihe von Unit-Tests besteht. In der Arbeit von [Che+21] wird daher ein detailliertes Verfahren zur Evaluierung von Code-LLMs beschrieben und die pass@k-Metrik vorgestellt, die auch in weiteren Arbeiten wie [Kul+19], [Li+23], [Guo+24] und [Roz+23] Anwendung findet.

pass@k-Metrik

In der Studie von [Che+21] hat man ein LLM verwendet, um mittels Docstrings Python-Funktionen zu generieren. Die Korrektheit der erzeugten Funktionen wurde anschließend mittels Unit-Tests automatisch überprüft. Zu diesem Zweck wurde ein Datensatz mit verschiedenen Programmierproblemen und den dazugehörigen Unit-Tests erstellt. Um die Wahrscheinlichkeit zu bestimmen, mit der ein Problem aus diesem Testsatz gelöst werden kann, wurden mehrere Codebeispiele mit dem gewählten Modell generiert und anschließend mit den vorbereiteten Unit-Tests getestet. Dabei wurde die pass@k-Metrik verwendet, um zu ermitteln, wie oft mindestens eine von k generierten Code-Varianten eine bestimmte Aufgabe erfolgreich löst.

Ergebnisse mit Vorsicht deuten

Wie in den Arbeiten [Zho+23] und [Aiy+23] erläutert, bestehen weiterhin Bedenken hinsichtlich der durch Benchmarks gewonnenen Ergebnisse. [Zho+23] beweist beispielsweise, dass die Leistung eines Modells bei einem Benchmark-Verfahren durch sogenannte Benchmark-Leakages erheblich verbessert werden kann:

> „The experimental results reveal that benchmark leakage can lead to an unfair boost in the evaluation performance of LLMs. Smaller LLMs

> [...] can be deliberately elevated to outperform 10x larger models on certain tasks."

Unter Benchmark-Leakage [Xu+24] versteht man die Verwendung von Datensätzen, die ursprünglich für die Bewertungssystemen verwendet werden, für das Training der eigenen Modelle. Wie bereits erwähnt, kann dies zu Leistungsverbesserungen in bestimmten Aufgabenbereichen führen, wodurch ein fairer Vergleich mit anderen Modellen nicht mehr möglich ist [Zho+23; Xu+24]. Es gibt aktuell auch keine Prüfung, ob derart unangemessene Methoden angewendet werden. [Xu+24] nennt hierfür mehrere Gründe:

> „(1) Cannot guarantee that the test data is leakage-free [...] (2) Difficult to determine the threshold score for leakage due to multiple influencing factors [...] (3) Unknown utilization of benchmarks [...] (4) Inaccessible model weights [...]."

3.4 Fuzz4All

Nachdem in Kapitel 2 die Grundlagen des Random Testings sowie die damit verbundenen Begriffe PBT und Fuzzing erläutert wurden und in diesem Kapitel ein grundlegendes Verständnis zu LLMs erworben wurde, soll abschließend ein Test-Tool namens „Fuzz4All" [Xia+23] vorgestellt werden, das die beiden Thematiken miteinander vereinte. Im Vergleich zu herkömmlichen Fuzzern, wie z. B. bei QuickCheck [CH00], werden bei Fuzz4All mithilfe eines LLMs die notwendigen Eingaben generiert, mit denen das SUT getestet wird. In [Xia+23] wird Fuzz4All somit als der erste Fuzzer vorgestellt, „[...] that is universal in the sense that it can target many different input languages and many different features of theses [sic] languages."

Fuzz4All-Prozess

Zunächst wird die Vorgehensweise betrachtet, mit der Fuzz4All das SUT testet. Der Prozess lässt sich, wie in Abbildung 6 dargestellt, in zwei Bereiche unterteilen: Autoprompting und Fuzzing Loop. Im ersten Schritt wird dem Test-Tool eine Eingabeaufforderung in Form von Dokumentationen, Beispiel-Code oder Spezifikationen übergeben. Wie in [Xia+23] erläutert, wären solche Dokumente zu umfangreich, um sie direkt als Prompt verwenden zu können. Da die manuelle Erstellung eines geeigneten Prompts ebenfalls als zu zeitaufwändig betrachtet wird, wurde ein Autoprompting-Schritt eingeführt. In diesem Schritt werden die bereitgestellten Dokumente an ein sogenanntes „Destillation"-LLM übergeben, das mehrere, möglichst unterschiedliche Prompts generiert. Mithilfe eines einfachen Fuzzing-Experiments werden schließlich die von dem „Destillation"-LLM

erzeugten Prompts bewertet. In diesem Ansatz wird jeder Prompt als Eingabe für das „Generation“-LLM verwendet, um mehrere Codeausschnitte pro Prompt zu erzeugen. Fuzz4All bewertet die generierten Codeschnipsel für jeden Prompt anhand einer Bewertungsfunktion. Diese Bewertungsfunktion überprüft im Wesentlichen, ob die generierten Codeschnipsel gültig sind und keine Fehlermeldungen erzeugen. Die Anzahl der gültigen Codeschnipsel ergibt schließlich den Score-Wert. Der Prompt mit dem höchsten Score wird dann als initialer Prompt für die Fuzzing Loop verwendet. Wie in [Xia+23] hervorgehoben, würde der gleiche Prompt über die Zeit zu vielen ähnlichen oder sogar identischen Modellausgaben führen. Um dies zu vermeiden, wurde in der Fuzzing Loop ein zusätzlicher Zwischenschritt eingeführt, der den Prompt nach jedem Durchlauf aktualisiert. Die vom LLM generierten „fuzzing inputs“ werden dann an das SUT weitergeleitet. Die dabei erzeugten Ergebnisse werden schließlich mit einem bereitgestellten Testorakel validiert. Als Beispiel für ein Testorakel wird hierbei die Überprüfung auf Systemabstürze genannt.

Fähigkeiten von Fuzz4All

Abschließend sollen noch die in der Arbeit von [Xia+23] beschriebenen Fähigkeiten und die damit verbundenen Vorteile zusammengefasst werden. Durch den Einsatz von LLMs, die meistens mit zahlreichen Programmiersprachen trainiert wurden, ermöglicht Fuzz4All eine umfassende Prüfung für eine Vielzahl von SUTs mit einem breiten Spektrum an verschiedenen Eingaben. Durch die Autoprompting-Komponente ist es möglich, ohne großen Aufwand automatisch Prompts zu generieren. Man müsse lediglich Dokumentationen oder ähnliche Informationen bereitstellen. Durch den Einsatz von LLMs kann sich das konstruierte Test-Tool zudem gemeinsam mit dem SUT weiterentwickeln. Als Beispiel werden hierfür neu hinzugefügte Features genannt. Es wäre lediglich erforderlich, die Dokumentation bereitzustellen, die sich auf diese Features beziehen. Durch den Fuzzing-Loop-

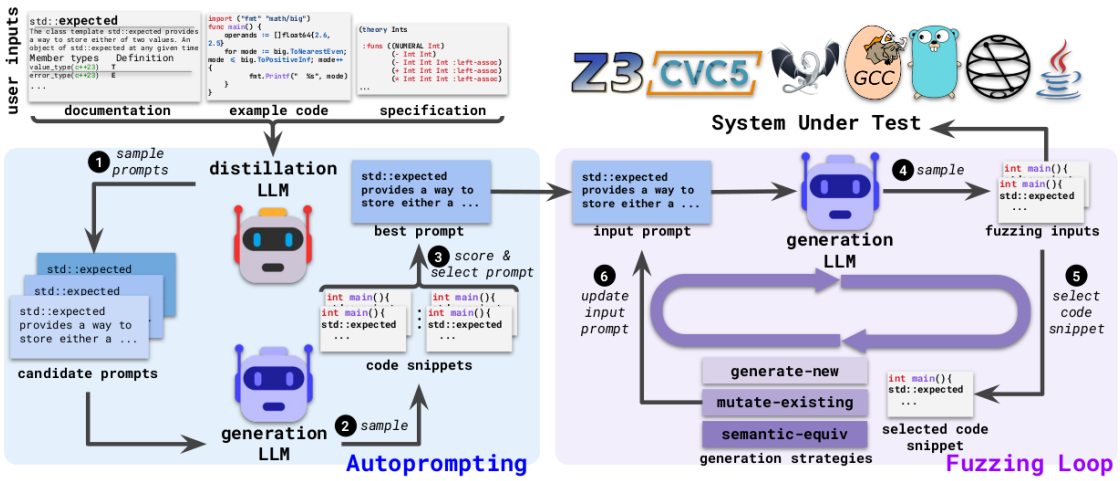


Abbildung 6: Fuzz4All-Ansatz [Xia+23]

Algorithmus können auch kontinuierlich neue Codebeispiele generiert werden, die zur Prüfung des SUTs verwendet werden können. Dabei ist es möglich verschiedene Strategien anzuwendet, um den Prompt nach jeder Durchführung zu modifizieren. Die Wirksamkeit des entwickelten Test-Tools wird durch die Entdeckung von insgesamt 98 Fehlern belegt, von denen 64 als unbekannt bestätigt wurden.

4 Konzeption des Prototyps

In diesem Kapitel wird zunächst ein Überblick über die einzelnen Systemkomponenten gegeben, um ein besseres Verständnis zu ermöglichen. Anschließend wird die OpenAPI-Spezifikation (OAS) und deren Struktur eingehend untersucht, wobei die notwendigen Einschränkungen und potenziellen Probleme für die späteren Durchführungen benannt werden. Im folgenden Abschnitt werden die Anforderungen an die LLM-Komponente definiert und eine klare Struktur für die Prompts entwickelt. Im letzten Abschnitt werden die für die späteren Untersuchungen notwendigen Kriterien vorgestellt und erläutert.

4.1 Systemkomponenten

In diesem Kapitel werden zunächst die einzelnen Komponenten der entwickelten Softwarelösung analysiert. Dabei werden die Abhängigkeiten zwischen den verschiedenen Komponenten sowie deren Interaktion miteinander untersucht. Ziel ist es, ein umfassendes Verständnis des Systemkontexts zu vermitteln. Hierfür soll die Abbildung 7 betrachtet werden. Die primäre Aufgabe des dargestellten Programms ist die Konstruktion eines Test-Tools zur Überprüfung des zu testenden Systems (SUT). Vor dem Start dieses Prozesses müssen drei wesentliche Dokumente von einer Person bereitgestellt werden: ein OAS-Dokument, eine Prompt-Vorlage und eine Konfigurationsdatei. Das OAS-Dokument beschreibt die zu testende Web-API und wird in mehreren Phasen des Hauptprogramms

Komponenten

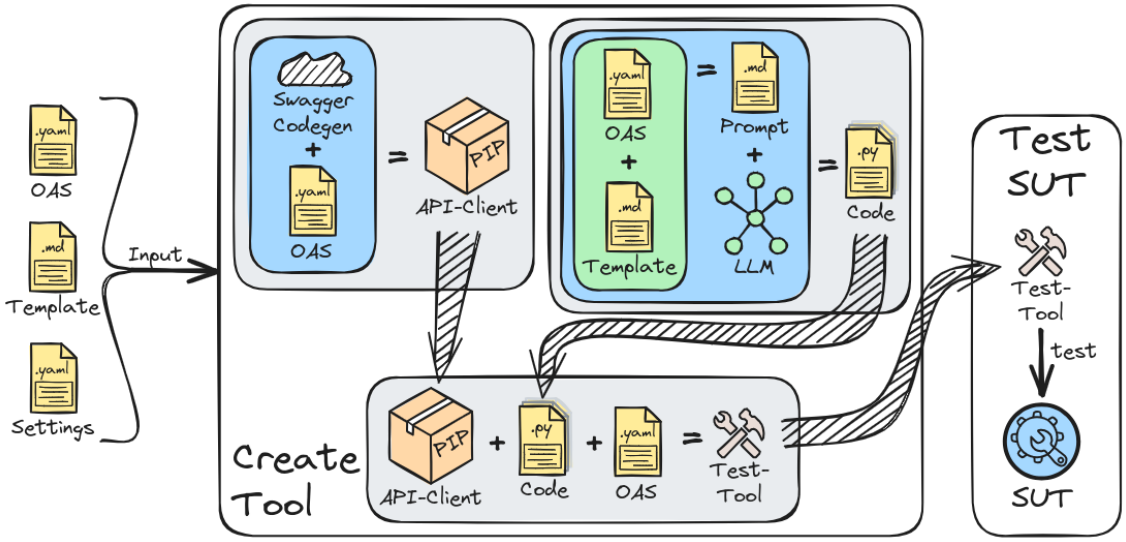


Abbildung 7: Kontext der Systemumgebung

benötigt. Das als Prompt-Vorlage bezeichnete Dokument enthält die Aufgabenbeschreibung sowie optionale Bestandteile wie Beispiele oder CoT-Anweisungen, die später an das LLM übergeben werden. Eine detailliertere Erläuterung des Inhalts dieses Dokuments befindet sich in Kapitel 4.3. Die Konfigurationsdatei spezifiziert die notwendigen Dateipfade zu den genannten Dokumenten und legt das Verhalten einzelner Komponenten fest. Eine ausführliche Beschreibung hierzu befindet sich in Kapitel 5.2. Die hier vorgestellte Swagger Codegen-Komponente [Gor+14] beinhaltet eine Engine, die mithilfe einer OAS einen API-Client generieren kann. Dabei besteht die Möglichkeit, die gewünschte Programmiersprache für den API-Client festzulegen. Da das zu konstruierende Test-Tool in Python implementiert wird, wird entsprechend auch ein API-Client für Python generiert. Das Ergebnis ist daher ein Python-Package, das anschließend über das Paketverwaltungsprogramm `pip` installiert werden kann. Die LLM-Komponente ist dafür verantwortlich, für alle Properties der OAS einen „Generator“ zu erstellen, der die notwendigen Werte für die Eingabeparameter generiert. Bei diesen Generatoren handelt es sich um Python-Dateien, die den dafür notwendigen Code beinhalten. Der Prompt, der dafür konstruiert und an das LLM weitergeleitet wird, besteht dabei aus der zuvor erwähnten Prompt-Vorlage und dem zusammengefassten Inhalt der OAS. Damit das Test-Tool automatisiert erstellt und anschließend verwendet werden kann, sind teilweise Komponenten erforderlich, die vom Hauptprogramm während des Prozesses generiert werden. Für die Konstruktion werden lediglich die vom LLM generierten Python-Dateien und die zu Beginn bereitgestellte OAS benötigt. Vor der Ausführung des Test-Tools muss der API-Client auf dem vorgesehenen System installiert werden.

4.2 OAS und Einschränkungen beim Testen des SUTs

Nach der Betrachtung des Systemkontexts soll in diesem Unterkapitel die Bedeutung der OAS erläutert werden. Dabei werden auch die in dieser Arbeit definierten Einschränkungen für die späteren Experimente beschrieben. Wie bereits in Kapitel 4.1 dargelegt wurde, wird die OAS von mehreren Komponenten verwendet. Während für die Swagger Codegen-Komponente keine weiteren Einschränkungen für die OAS definiert werden müssen, verhält es sich bei der LLM-Komponente anders. Damit später auch Funktionen generiert werden, die ausschließlich gültige Werte erzeugen, müssen entsprechende Eigenschaften definiert und im Prompt hinterlegt werden. Die erforderlichen Eigenschaften für die Eingabeparameter können hierbei aus der OAS extrahiert werden.

OAS-Struktur

Damit die in der OAS hinterlegten Eigenschaften der Eingabeparameter extrahiert werden können, muss zunächst die Struktur der OAS betrachtet werden. Die Struktur der OAS kann in folgende Bestandteile unterteilt werden: Metadaten, Server, Pfade, Komponenten und Sicherheit. Die Metadaten-Komponente besteht aus zwei Abschnitten, in denen sowohl die Version der OAS als auch allgemeine Informationen über die API spezifiziert werden. In der Server-Komponente können mehrere API-Server und ihre Basis-URLs hinterlegt werden. Die Pfad-Komponente enthält die verschiedenen Endpunkte und ihre zugehörigen HTTP-Methoden, die über die API bereitgestellt werden. Ein Endpunkt mit der entsprechenden HTTP-Methode kann aus bis zu drei verschiedenen Abschnitten bestehen. Im Response-Abschnitt werden die verschiedenen Status-Codes beschrieben und muss bei allen Endpunkten und ihren HTTP-Methoden enthalten sein. Wenn die URL des Endpunkts zusätzlich Pfad- oder Query-Parameter enthält, müssen diese in einem Abschnitt für Parameter spezifiziert werden. Der Request-Body wird definiert, wenn über die Operation noch Daten über den HTTP-Body gesendet werden können. Es kann vorkommen, dass verschiedene API-Operationen ähnliche Parameter oder einen ähnlichen HTTP-Body spezifizieren. Um solche Duplikationen in der OAS zu vermeiden, wird ein Abschnitt in der OAS-Struktur bereitgestellt, in dem globale Definitionen erstellt und referenziert werden können. Schließlich gibt es auch eine Sicherheitskomponente, in der Sicherheitsmechanismen für die Autorisierung und Authentifizierung definiert werden können.

Einschränkungen

Da später sogenannte Patterns für String-Datentypen verwendet werden sollen, ist es erforderlich, dass eine OAS der Version 3 verwendet wird. Für die Durchführung der Tests werden zudem keine Sicherheitsmechanismen betrachtet und daher auch nicht im SUT implementiert. Weiterhin werden auch keine Referenzen auf globale Definitionen verwendet, damit die automatisierte Analyse der OAS möglichst einfach umgesetzt werden kann. Die Abschnitte Metadaten, Server und Pfade sind jedoch erforderlich und müssen daher spezifiziert werden. Für die Extrahierung der Eingabeparameter und ihrer Eigenschaften sind jedoch nur die Abschnitte innerhalb der Komponente von Bedeutung, in der die verschiedenen Endpunkte mit deren Operationen spezifiziert werden. Verfügt eine Operation über einen Parameter- oder Request-Body-Abschnitt, werden die dort spezifizierten Eingabeparameter mit deren Eigenschaften später entnommen.

Extraktion der Daten

Wie man bereits in Kapitel 3.4 erfahren hat, ist es allerdings nicht sonderlich ratsam komplette Dokumente unverarbeit an dass LLM zu übergeben, da diese meistens zu umfangreich sind. Die meisten Modelle haben nur eine begrenzte Kontextlänge, und je umfangreicher die verwendete OAS ist, desto schneller

```
- {variable1: {type:string,pattern:[\s\S]{0,20}}}
- {variable2: {type:string,pattern:([A-Za-z0-9]{5}-){2}[A-Za-z0-9]{5},
  ↪  example:uDqmG-jR3Bp-ad3FE}}
- {variable3: {type:string,format:date-time}}
- {variable4: {type:string,format:byte,pattern:[A-Za-z0-9+/]{0,60}={0,3},
  ↪  example:U3dhZ2dlciByb2Nrcw==}}
- {variable5: {type:number,format:float,minimum:-1.23,maximum:534.74523}}
```

Listing 3: Mehrere Eingabeparameter zu einem Property

kann die maximale Kontextlänge erreicht werden, wenn diese unverändert im Prompt hinterlegt wird. Darüber hinaus werden in dieser Arbeit Modelle eingesetzt, die bereits für die Generierung von Code vortrainiert und folglich auf eine generalisierte Problemstellung ausgelegt sind. Um die speziell für diese Arbeit definierte Aufgabe zu lösen, sind daher auch noch zusätzliche Anweisungen neben der OAS erforderlich, damit das LLM angemessen angeleitet werden kann. Um die Anweisungen, die an den Transformer übergeben werden, möglichst einfach zu halten, werden die aus der OAS extrahierten Informationen nicht nur zusammengefasst, sondern auch in mehrere Bündel zerlegt. Anstatt alle Properties und deren Eingabeparameter mitsamt ihrer Eigenschaften in einem einzigen Bündel zusammenzufassen, wird ein Bündel pro Property konstruiert. Insbesondere bei einer OAS mit vielen Endpunkten und entsprechend vielen Parametern kann dadurch der Umfang der Prompts nochmals deutlich reduziert werden. Ein Bündel beinhaltet hierbei eine Liste an Eingabeparameter mit deren Eigenschaften. Bevor diese Informationen extrahiert werden können, ist es notwendig, zunächst eine geeignete Datenstruktur zu konstruieren, in dem diese Informationen hinterlegt werden sollen. Aufgrund der Verwendung von LLMs, die mit Python-Code trainiert wurden, wurde eine Datenstruktur gewählt, die in der Programmiersprache Python existiert. Die aus der OAS extrahierten Daten werden daher in Form eines Dictionaries hinterlegt. Ein entsprechendes Beispiel eines solchen Bündels wird in Listing 3 veranschaulicht.

Datentypen & Schlüsselwörter

Während Anpassungen am Prompt dazu dienen die Ausgabe des LLMs immer weiter zu optimieren, sollte mit den Veränderungen an der OAS untersucht werden, wie gut das LLM mit unterschiedlich komplexen Datentypen umgehen kann. OpenAPI definiert hierbei sechs Basis-Datentypen: *string*, *number*, *integer*, *boolean*, *array* und *object*. Weil die beiden Datentypen *array* und *object* eine Menge von gleichen bzw. unterschiedlichen Datentypen darstellen, habe ich meine Untersuchungen auf die Datentypen *string*, *number*, *integer* und *boolean* eingeschränkt. Zusätzlich zu diesen Basis-Datentypen können auch noch typspezifische Schlüsselwörter verwendet werden, um Eigenschaften dieser Datentypen

Datentyp	Schlüsselwörter
Number &	minimum
Integer	maximum
	enum
	format

(a) Nummerische Datentypen

Datentyp	Schlüsselwörter
String	pattern
	enum
	format
	example

(b) String Datentypen

Datentyp	Format	Beschreibung
Number	-	Beliebige Zahlen.
	float	Fließkommazahlen.
	double	Fließkommazahlen mit doppelter Genauigkeit.
Integer	-	Ganzzahlige Zahlen
	int32	32-Bit-Ganzzahlen mit Vorzeichen.
	int64	64-Bit-Ganzzahlen mit Vorzeichen.
String*	date	Datumsschreibweise gemäß RFC 3339, Abschnitt 5.6.
	date-time	Datums-Zeit-Notation, wie in RFC 3339, Abschnitt 5.6 definiert.
	password	Hinweis an die UI, die Eingabe zu maskieren.
	byte	base64-kodierte Zeichen.
	binary	binäre Daten, die zur Beschreibung von Dateien verwendet werden.

(c) Formatmodifikatoren

* Ein optionaler Formatmodifikator dient als Hinweis auf den Inhalt und das Format der Zeichenkette. Format ist hier jedoch ein offener Wert, so dass beliebige Formate verwendet werden können, auch solche, die nicht in der OpenAPI-Spezifikation definiert sind.

Tabelle 3: Basis-Datentypen und ihre typspezifischen Schlüsselwörter [Swa24]

zu beschreiben. Bei den Versuchsdurchführungen werden allerdings nicht alle Schlüsselwörter für jeden Datentypen betrachtet. Daher ist Tabelle 3 (a & b) lediglich eine Zusammenfassung aller Datentypen, die in den Versuchsdurchführungen mit einbezogen werden. Mit *minimum* und *maximum* wird eine untere und obere Grenze definiert. Mit dem Schlüsselwort *pattern* kann man eine Vorlage für einen regulären Ausdruck für die Zeichenfolge eines String-Datentypens definieren. Mit *enum* wird eine Menge an möglichen Werten definiert, die der Eingabeparameter annehmen kann. Durch das Schlüsselwort *example* kann zusätzlich auch noch ein Beispiel angegeben werden. Der *format*-Modifikator dient als Hinweis auf den Inhalt und das Format des Datentypens. Mögliche Werte für den Formatmodifikator werden in Tabelle 3 (c) zusammengefasst und erklärt.

Abdeckungsproblem

Bei der Analyse der Operationen und deren Eingabeparametern wird ausschließlich die OAS betrachtet. Diese gibt Aufschluss über die von der API bereitgestell-

ten Operationen und ermöglicht es die Eingabeparameter mit deren Eigenschaften den entsprechenden Operationen zuzuordnen. Die Interna des SUTs sind allerdings verborgen. Mit den Informationen aus der OAS können mit dem Test-Tool durchaus valide Anfragen an das SUT gesendet werden, aber aufgrund der Erzeugung von zufälligen Werten für die Eingabeparameter ist es nicht garantiert, dass alle Statuscodes bei einer Testdurchführung abgedeckt werden können [AGP19]. Als Beispiel soll hierfür eine Ressource betrachtet werden, die als ID einen alphanummerischen String mit 32 variablen Zeichen verwenden. In der OAS wird eine GET Operation mit den notwendigen Parametern und Eigenschaften definiert, damit eine valide Anfrage an die API erzeugt werden kann. Sind in der Datenbank nur 10 Einträge vorhanden, so ist die Wahrscheinlichkeit einen zufälligen Wert mit einer existierenden ID zu erzeugen sehr gering.

4.3 Modellanforderungen und -anweisungen

Für die Umsetzung wird das Transformer-Package von Huggingface mit PyTorch als Framework verwendet. Damit wird die Ausführung der Inferenz auf die CPU oder die GPU eingeschränkt. Um ein geeignetes Modell für die beschriebene Aufgabe zu finden, sollten zunächst die erforderlichen Anforderungen an das Modell klar definiert werden. Auf Basis der definierten Anforderungen können anschließend passende Modelle ausgewählt und miteinander verglichen werden. Weiterhin muss auch ein zum Modell passender Prompt konstruiert werden, der die notwendigen Anweisungen und Informationen enthält. Neben der Auswahl des Modells und der Erstellung eines Prompts ist auch die Auswahl einer geeigneten Dekodierstrategie erforderlich. Die Dekodierstrategie bestimmt hierbei die Methode, nach der das LLM seine Antworten generiert. Dabei wird festgelegt, nach welchen Kriterien der Decoder den nächsten Token auswählt. Zunächst werden die für das Modell definierten Anforderungen betrachtet.

Anforderungen an das Modell

Modellgröße Für die Inferenz wird eine GPU mit 40 GB RAM verwendet. Damit die Ausführungszeit der Inferenz möglichst kurz ist, sollen ausschließlich Modelle verwendet werden, die vollständig von der Grafikkarte geladen werden können. In diesem Zusammenhang können auch Quantisierungsverfahren angewendet werden, um die Präzision des Modells auf 8-Bit oder sogar auf 4-Bit zu reduzieren.

Aufgabenbereich Wie bereits erwähnt, soll das LLM Code generieren, welches zufällige Werte für die Eingabeparameter eines Properties erzeugt. Diese

Werte müssen dabei im defnierten Bereich sein, wie in der OAS beschrieben wird. Dafür werden Anweisungen in Form von Text übermittelt, woraufhin das Modell eine entsprechende Antwort in Textformat generiert. Es wird daher ein Modell gesucht, das sowohl für die Verarbeitung natürlicher Sprachen als auch für die spezialisierte Generierung von Code geeignet ist.

Architekturtyp Grundsätzlich kann man zwischen drei Architekturtypen unterscheiden. Zum einen die Encoder-Decoder Architektur, wie sie in Kapitel 3.1 beschrieben wird. Es gibt allerdings auch Architekturen die entweder nur den Encoder oder nur den Decoder verwenden. Für die spätere Umsetzung empfiehlt es sich ausschließlich Modelle zu betrachten, die entweder eine Encoder-Decoder oder eine Decoder-Only Architektur verwenden.

Vortrainierte Modelle Das Training eines Modells von Grund auf bietet mehrere Vorteile. Allerdings ist dieser Prozess zeitaufwendig, da zunächst ein geeignetes Datenset erstellt und zahlreiche Tests durchgeführt werden müssen, um das gewünschte Ergebnis zu erzielen. Die für das Training verwendeten Daten beeinflussen dabei das Verhalten des LLMs und sind entscheidend dafür, ob sich das trainierte LLM für die vorgesehene Aufgabe auch eignet. Angesichts des damit verbundenen hohen Zeitaufwands werden für die späteren Untersuchungen ausschließlich vortrainierte Modelle verwendet. Zudem soll darauf geachtet werden, dass die verwendeten Modelle auch mit Datensets trainiert wurden, die Python Code beinhalten.

Leistung Die Leistung bzw. die Fähigkeiten Aufgaben zu lösen wird meistens über sogenannte Benchmarks gemessen. Wie bereits in Kapitel 3.2 und 3.3 hervorgeht, kann dies jedoch auch zu Missinterpretationen führen. Daher sollte die Eignung eines Modells für die vorgesehene Aufgabe anhand einer eigenen und zur Problemstellung passenden Evaluation überprüft werden.

Open Source Es sollen ausschließlich quelloffene Modelle verwendet werden.

Konstruktion der Prompts

Die verwendeten Anweisungen, auch bekannt als Prompts, haben einen großen Einfluss auf die vom LLM generierten Ergebnisse. Einen geeigneten Prompt zu konstruieren, der die notwendigen Eigenschaften besitzt, damit das verwendete LLM Lösungen generieren kann, die den festgelegten Anforderungen entsprechen, ist oft mit einer experimentellen Herangehensweise verbunden. Für die späteren Experimente und auch für eine bessere Visualisierung hat man eine entsprechende Struktur für die Prompts definiert, die in Abbildung 8 auch dargestellt wird. Hierbei handelt es sich zunächst nur um eine Vorlage, die erst in Verbindung mit den in Kapitel 4.2 erwähnten Informationen aus der OAS den

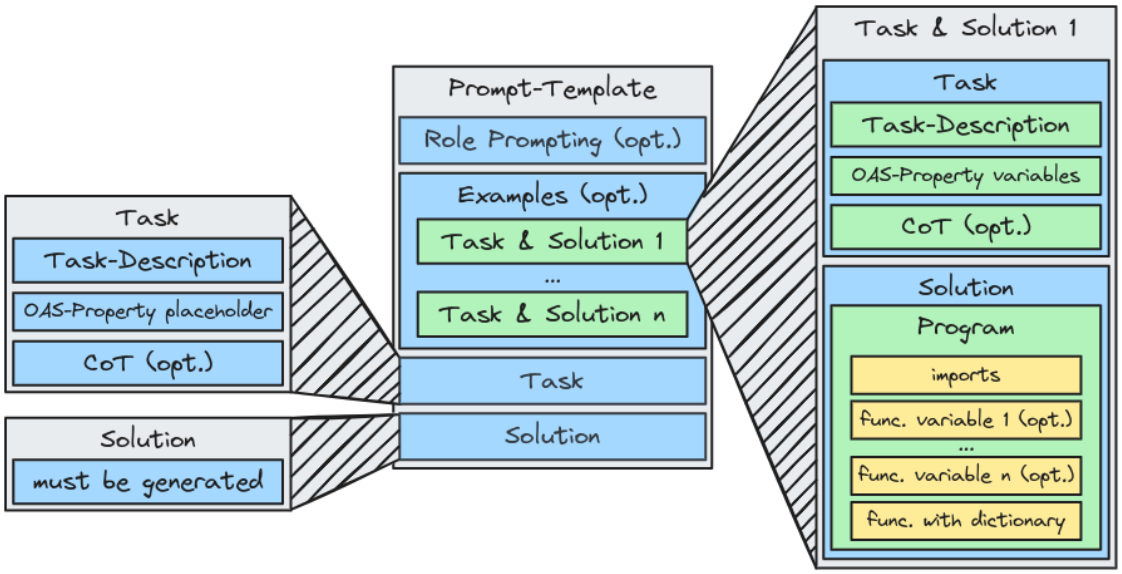


Abbildung 8: Struktur der Prompt-Vorlage

vollständigen Prompt erzeugt, der dann an den Transformator des LLMs übergeben wird. Diese Vorlage kann in die folgenden drei Teile unterteilt werden: Role Prompting, Beispiele und Endgültige Aufgabe mit der zu generierenden Lösung. Im ersten Teil kann dem LLM eine geeignete Rolle zugewiesen werden. Im Anschluss kann eine beliebige Anzahle an Beispielen vorgegeben werden. Dabei besteht ein Beispiel jeweils aus einer Aufgabe und der zur Aufgabe gewünschten Lösung. Die Aufgabe umfasst hierbei eine Aufgabenbeschreibung, die Eingabeparameter zu einem Property und deren Eigenschaften und optional noch eine *Chain-of-though*-Anweisung. Abgeschlossen wird das Beispiel mit der zur Aufgabe passenden Musterlösung in Form eines Programms. Im letzten Abschnitt wird wie bei den Beispielen wieder eine Aufgabe beschrieben. Allerdings mit dem Unterschied, dass für die Eingabeparameter des Properties stattdessen ein Platzhalter verwendet wird. Die Lösung wird dann schließlich vom LLM generiert. Der Platzhalter wird dann bei der Durchführung der Tests mit den aus der OAS herrausgefilterten Informationen über die Eingabeparameter eines Properties ersetzt.

4.4 Kriterien für die Evaluation der Modellausgaben

Evaluationsmodell

Bevor man mit den eigentlichen Tests für die Optimierung der Modellausgaben beginnen kann, ist es notwendig ein Bewertungsmodell zu konstruieren. Mithilfe eines geeigneten Bewertungsmodells kann man anhand von vorher definierten Kriterien die vom LLM erzeugten Ausgaben bewerten indem Punkte zugeteilt wer-

den. Dies ermöglicht es die verschiedenen Ergebnisse dann auch untereinander besser vergleichen zu können. Für die Evaluierung hat man sich für eine qualitative Vorgehensweise entschieden. Wie in Kapitel 3.2 beschrieben, wurde dies auch als subjektive Bewertung bezeichnet, da die Ergebnisse von einer Einzelperson bewertet werden und sich die Bewertungsergebnisse je nach den definierten Kriterien von Person zu Person unterscheiden können. Obwohl der Zeitaufwand für die Bewertung erheblich größer ist, wird durch diese Vorgehensweise erwartet, dass die Ursachen verschiedener Probleme besser identifiziert werden können.

Kriterien

Um die Auswirkungen solcher Schwankungen, die durch eine solch subjektive Bewertung entstehen können, dennoch zu minimieren, wurden Kriterien festgelegt, die entweder mit „erfüllt“ (1) oder „nicht erfüllt“ (0) beantwortet werden. In Tabelle 4 werden die für das Bewertungsmodell verwendeten Kriterien zusammengefasst. Dabei wird auch definiert, wann ein Kriterium als „erfüllt“ bzw. „nicht erfüllt“ betrachtet werden kann. Mit den Kriterien „Ausführbarkeit“ und „Zufälligkeit“ soll überprüft werden, ob das vom LLM generierte Programm fehlerfrei ist und auch Werte zur Verfügung stellt, die sich im definierten Bereich befinden.

Damit die Einbindung der erzeugten Teilprogramme in den Gesamtprozess möglichst einfach ist, muss auch eine geeignete Datenstruktur des Rückgabewerts bestimmt werden, der über alle notwendigen Eingabewerte verfügt. Dafür können zum einen die in Python vorhandenen Listen und zum anderen die Dictionaries als mögliche Datenstrukturen verwendet werden. Für die spätere Durchführung wählte ich die Speicherung der Daten in Form von Key-Value-Paaren und entschied mich daher für die Verwendung einer Dictionary-Datenstruktur. Zudem erschien es als sinnvoll, dass für die Funktion, welche die Werte für die Eingabeparameter zu Verfügung stellt, stets derselbe Funktionsname verwendet wird, damit diese eindeutig identifiziert werden kann. In Tabelle 4 werden diese Kriterien mit „Korrekter Rückgabewert“ und „Korrekter Funktionsname“ bezeichnet.

Kriterium	Erfüllt?	Beschreibung
Ausführbarkeit (exe)	1	Die vom LLM generierten Funktionen können, ohne das noch was modifiziert werden muss, ausgeführt werden. Dabei spielt die Korrektheit oder Sinnhaftigkeit des Ergebnisses keine Rolle.
	0	Mindestens eine Funktion kann aufgrund eines beliebigen Fehlers nicht ausgeführt werden.
Zufälligkeit (rnd)	1	Alle von dem LLM generierten Funktionen, die für die Generierung von Werten für die Eingabeparameter verantwortlich sind, haben die Fähigkeit, zufällige Werte zu erzeugen, die gemäß der OAS im definierten Bereich liegen. Die zufälligkeit kann dabei auch erfüllt sein, wenn die Funktion nicht ausgeführt werden kann. Z. B. wenn ein benötigtes Package nicht importiert ist.
	0	Es werden von mindestens einer Funktion Werte erzeugt, die sich nicht im definierten Bereich befinden oder diesen Bereich nicht vollständig abdecken.
Korrekter Rückgabewert (crv)	1	Der Rückgabewert ist ein Dictionary, das eine Anzahl von Einträgen aufweist, die der Anzahl der in der Aufgabenbeschreibung definierten Variablen entspricht. Die Schlüsselbezeichnungen im Dictionary entsprechen den Bezeichnungen der Variablen, und jedem Schlüssel ist ein zur Variable passender Wert zugeordnet.
	0	Wenn entweder kein Dictionary zurückgegeben wird oder das zurückgegebene Dictionary nicht der Definition entspricht, wird dies als Fehler betrachtet. Selbst wenn ein Dictionary alle Bedingungen erfüllt, aber nicht zurückgegeben wird, wird dies ebenfalls als Fehler gewertet.
Korrekter Funktionsname (cfn)	1	Die Funktion, die das Dictionary mit den Werten für die Eingabeparameter enthält, ist einem Funktionsnamen zugeordnet, der vom Tester vorher festgelegt wird.
	0	Ein alternativer Funktionsname wird zugewiesen, der nicht vom Tester bestimmt wurde.

Tabelle 4: Kriterien und deren Definitionen

5 Implementierung des Prototyps

Dieses Kapitel behandelt die Implementierung der konzeptionierten Softwarelösung. Zunächst wird der gesamte Prozessablauf bis zur Erstellung des Test-Tools beschrieben. Anschließend wird die Projektstruktur des zu konstruierenden Programms vorgestellt, wobei die einzelnen Bestandteile, wie z. B. die Device-Map, im Detail erläutert werden. Nach der Beschreibung des Programms zur Erzeugung des Test-Tools werden das Test-Tool selbst und die damit verbundenen Prozesse betrachtet. Abschließend wird noch ein kurzer Überblick über die zu testende Web-API gegeben.

5.1 Prozessablauf

Das Programm, welches das Test-Tool erstellt, wurde so konzipiert, dass es in mehreren Phasen betrachtet werden kann. Abbildung 9 zeigt den groben Ablauf des gesamten Prozesses. Für eine detaillierte Betrachtung der als Phasen I bis III gekennzeichneten Zwischenschritte verweise ich auf die im Anhang bereitgestellten Flussdiagramme (siehe Anlage F: Programmablaufplan).

Konfigurationen laden

❶ Bevor der eigentliche Prozess gestartet werden kann, müssen zunächst die für die einzelnen Phasen erforderlichen Einstellungen aus der Konfigurationsdatei geladen werden. Der Pfad zu dieser Datei wird dabei über die Konsole angegeben.

API-Client erstellen

❷ In Phase I wird der API-Client für das Test-Tool erstellt. Ob diese Phase ausgeführt wird, hängt von zwei Kriterien ab: Ein gesetztes Flag und ein in der Konfigurationsdatei hinterlegter Hashwert des OAS-Dokuments, der überprüft, ob eine andere OAS-Version verwendet wird. Wird eine andere oder aktualisierte OAS-Version verwendet, wird der Prozess zur Aktualisierung des API-Clients ausgeführt. Dazu wird eine API-Anfrage an Swagger Codegen gesendet, die über den

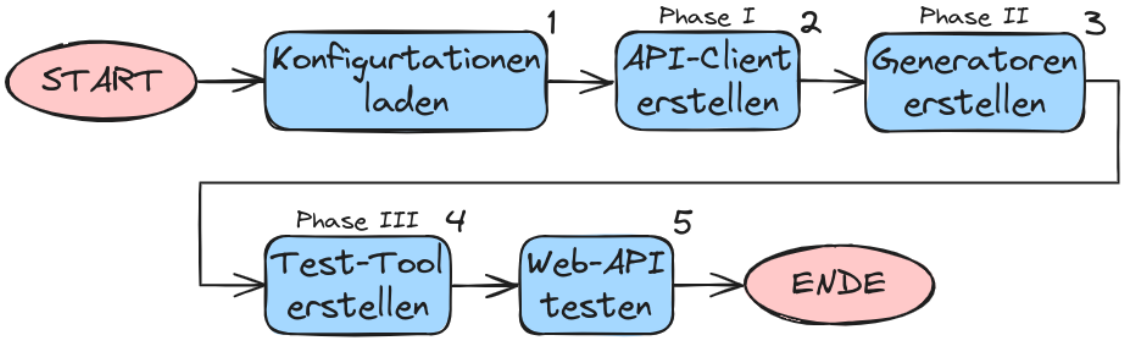


Abbildung 9: Überblick zum Gesamtprozess

HTTP-Body die OAS, die gewünschte Programmiersprache, den API-Typ (Client oder Server) und die verwendete OAS-Version übermittelt. Swagger Codegen erzeugt die notwendigen Dateien und sendet sie als Zip-Datei zurück, die anschließend entpackt und installiert wird. Der Vorgang endet mit der Aktualisierung des Hashwerts. ❸ Phase II nutzt die LLM-Komponente zur Generierung der für das Test-Tool erforderlichen Generatoren. Auch hier wird mittels eines Flags entschieden, ob diese Phase ausgeführt werden soll. Bevor das ausgewählte Modell auf die GPU geladen werden kann, muss die Person sich mit ihren Anmeldedaten bei Huggingface authentifizieren. Anschließend wird das Dokument verwendet, um die einzelnen Properties zu identifizieren und in einer Liste zusammenzufassen. Wie in Kapitel 4.2 erläutert, ist dieser Schritt notwendig, um Bündel zu erzeugen, welche die erforderlichen Informationen der einzelnen Properties enthalten. Die Bündel werden nach und nach mit der zuvor bereitgestellten Prompt-Vorlage kombiniert, um den in Kapitel 4.3 beschriebenen Prompt zu erstellen, der an das LLM übergeben wird. Die LLM-Komponente generiert daraufhin den entsprechenden Python-Code, der anschließend als Datei abgespeichert wird. Anschließend erfolgen noch zwei optionale Überprüfungen. Zunächst wird geprüft, ob ein Resampling erforderlich ist. Wenn nicht ausreichend Tokens bzw. Zeichen generiert wurden und das in Kapitel 4.4 definierte Kriterium für den Funktionsnamen nicht erfüllt ist, wird der Vorgang wiederholt und die vorherige Datei überschrieben. Die zweite Überprüfung stellt sicher, dass im Code verwendete Packages auch korrekt importiert wurden und ergänzt diese bei Bedarf. Der Vorgang wird solange wiederholt, bis alle Properties einen eigenen Generator in Form einer Python-Datei haben. ❹ In Phase III wird das Test-Tool basierend auf den von einer Person und den von den vorherigen Phasen bereitgestellten Komponenten erstellt. ❺ Es handelt sich hierbei um ein eigenständiges Programm, das verwendet werden kann, um die Web-API zu testen. Eine umfassende Erläuterung zur Projektstruktur und Funktionsweise des Test-Tools ist in Kapitel 5.3 zu finden.

Generatoren erstellen

Test-Tool erstellen

Web-API testen

5.2 Hauptprogramm

In diesem Kapitel soll die für das Projekt verwendete Struktur vorgestellt werden, die auch nochmal in Listing 4 dargestellt wird. Die für die Umsetzung konstruierte Projektstruktur besteht aus den folgenden Bestandteilen: Konfigurationsdateien, OAS-Dateien, Vorlagen, Hauptprogramm und den vom Programm erzeugten Ausgaben.

```
Project Structure
|-configs
| |-config-files
| |-device-maps
|-Fuzzer
| |-model
| |-util
| |-main.py
|-OAS
|-templates
|-output
| |-group
| | |-date-time
| | | |-data
| | | |-generated-code
| | | |-generated-prompts
| | | |-generated-test-tool
| | | |-logger.md
| | | |-system-information.md
| |-api-client
```

Listing 4: Projektstruktur vom Hauptprogramm

Konfigurationsdatei

In diesem Projekt unterscheidet man zwischen zwei verschiedenen Konfigurationsdateien. Zunächst wird die Konfigurationsdatei im Ordner `config-files` betrachtet. Diese Datei ist in mehrere Abschnitte unterteilt, einschließlich allgemeiner Konfigurationen, Modellkonfigurationen und spezifischer Konfigurationen für die einzelnen Phasen, die das Programm durchläuft. Im Abschnitt für die allgemeinen Konfigurationen wird der Ordner für die vom Programm erzeugten Ausgaben spezifiziert, sowie der Pfad zur gewählten OAS-Datei. Damit die Ausgaben auch gruppiert werden können, wird noch die Möglichkeit gegeben einen Gruppennamen zuzuteilen. Im Abschnitt für die Modellkonfigurationen werden zum einen der Checkpoint des verwendeten Modells als auch sämtliche Einstellungen spezifiziert, die für die Instanziierung des Modells erforderlich sind. Dabei kann man frei entscheiden, welches Modell verwendet werden soll, sofern es sich um ein Modell für kausale Sprachmodellierung handelt und die Architektur von der Transformers-Bibliothek unterstützt wird. Zusätzlich können noch folgende Einstellungen konfiguriert werden: Modellparameter, Modellpräzision, Device-Map, maximale Anzahl neuer Tokens, Tokens zur Definition des Sequenzendes sowie Resampling-Bedingungen. Der Abschnitt für die erste Phase, also das Erstellen des API-Clients, beinhaltet ein Feld mit der Bezeichnung `oas-checksum`. Dabei handelt es sich um den zuvor schon erläuterten Hashwert, der verwendet wird, um Veränderungen an der OAS zu identifizieren. Im Feld `base-url` wird die Basis-URL der Swagger Codegen API angegeben. Der Abschnitt für die Phase II spezifiziert zum einen den Pfad zur Prompt-Vorlage und zum anderen ein Feld mit

Allgemein

Modell

Phasen

```
{
  "transformer.wte": 0,
  "lm_head": 0,
  "transformer.wpe": 0,
  "transformer.drop": 0,
  "transformer.h.0": 0,
...
  "transformer.h.11": 0,
  "transformer.h.12": "cpu",
...
  "transformer.h.39": "cpu",
  "transformer.ln_f": "disk"
}
```

Listing 5: Device-Map für das Modell bigcode/starcoder

der Bezeichnung `loops`. Mit dem `loops`-Feld wird festgelegt, wieviele Stichproben pro Property erstellt werden sollen. Dafür wird für gewöhnlich der Parameter `batch-size` der Generatorfunktion angepasst. Allerdings können dann nicht die die Ausführungszeiten für die einzelnen Stichproben ermittelt werden, weshalb eine Alternative Umsetzung mit der Variable `loops` implementiert wurde. In der letzten Phase kann nur die Anzahl an Tests spezifiziert werden, die dass konstruierte Test-Tool ausführen soll. Alle Phasen verfügen zudem noch über ein `execute`-Flag, welches darüber entscheidet, ob die Phase ausgeführt werden soll oder nicht.

Device-Map

Bei der anderen Konfigurationsdatei handelt es sich um die Device-Map. Ein Modell kann über mehrere Layer verfügen, die auch beliebig groß sein können. Die an den Transformer übergebene Eingabe muss dabei diese Layer während der Inferenz durchlaufen. Für diese Operationen werden die Schichten normalerweise vollständig in den GPU-Speicher geladen. Allerdings ist dies nicht immer möglich. Entweder fehlt es an einer Grafikkarte mit ausreichend Speicher oder das Modell ist generell zu groß, um vollständig geladen zu werden. Die Idee hinter einer benutzerdefinierten Gerätezuordnung besteht darin, dass die verschiedenen Schichten des Modells auf unterschiedliche Hardware-Komponenten verteilt werden können. Wenn beispielsweise mehrere Grafikkarten vorhanden sind, können die Modellschichten zwischen den beiden Hardwarekomponenten aufgeteilt werden. Es ist auch möglich, Schichten im Systemspeicher des Prozessors zu platzieren. Wenn dies nicht ausreicht, können Schichten auch auf dem Festplattenspeicher abgelegt werden. Wenn die Eingabe ein Layer durchlaufen muss, der sich nicht auf der GPU befindet, wird das erforderliche Layer von der CPU auf die GPU oder von der Festplatte auf die CPU und dann auf die GPU geladen. Nachdem die Eingabe die Schicht durchlaufen hat, wird dieses Layer aus dem GPU-

Speicher wieder entfernt. Je nachdem, wie die Layer schließlich zwischen den verschiedenen Hardwarekomponenten aufgeteilt werden, kann dies zu deutlich längeren Inferenzzeiten führen. Dennoch ermöglicht diese Methode das Laden von Modellen beliebiger Größe, solange das größte Layer von der GPU geladen werden kann. Im vorliegenden Listing 5 wird eine exemplarische Darstellung einer potenziellen Zuordnung von Geräten präsentiert. Im Falle eines Systems mit mehreren Grafikkarten können diese GPUs durch ganzzahlige Werte wie 0, 1 usw. referenziert werden.

OAS-Dateien & Prompt-Vorlagen

Die OAS-Dateien und Prompt-Vorlagen werden in seperaten Ordnern hinterlegt und können über die Konfigurationsdatei entsprechend referenziert werden. Diese Dateien müssen vom Tester erstellt und zur Verfügung gestellt werden. Bei der Konstruktion der OAS ist es erforderlich, die in Kapitel 4.2 aufgeführten Einschränkungen zu beachten. Für die Gestaltung der Prompt-Vorlagen sollten die in Kapitel 4.3 entwickelten Designvorschläge berücksichtigt werden.

Fuzzer

Die in Abbildung 4 hervorgehobenen Zeilen 5 bis 8 enthalten die Dateien des eigentlichen Programms zur Erstellung des Test-Tools. Im `model`-Ordner (Zeile 6) befindet sich eine Python-Datei, die über alle notwendigen Operationen zur LLM-Komponente verfügt. Dazu gehören unter anderem die Authentifizierungsfunktion, die Funktion zur Erstellung des Prompts sowie die Inferenzfunktion, welche die Tokens und somit die Antwort des Modells generiert. Auch die für die Inferenzfunktion erforderliche `«EndOfFunctionCriteria»`-Klasse wird hier definiert. Diese überprüft nach jedem generierten Token, ob in der erzeugten Sequenz ein bestimmter Token oder eine bestimmte Sequenz von Tokens enthalten ist. Falls einer dieser End-of-Sequence (EOS)-Strings in der erzeugten Sequenz vorkommt, wird der Vorgang abgeschlossen.

Der `util`-Ordner enthält mehrere Dateien, die im Folgenden schrittweise untersucht werden. Zunächst wird die Datei `Logger.py` betrachtet. Während der Programmausführung sollen die Ereignisse sowohl in der Konsole angezeigt als auch in einer Datei protokolliert werden. Zusätzlich sollen die Systeminformationen in einer separaten Datei gespeichert werden. Die `Logger.py`-Datei stellt hierfür die erforderlichen Funktionen bereit. Die Datei `dataCollector.py` hingegen bietet Funktionen an, um Tabellen zu erstellen und weitgehend automatisch mit Daten zu füllen, die später für die Auswertung benötigt werden. Die Datei `codegenApi.py` beinhaltet eine Funktion, mit der die API-Anfrage an Swagger Codegen erstellt und übermittelt werden kann. Diese Funktion speichert auch die erhaltene Zip-Datei auf dem System und entpackt deren Inhalt automatisch. Die Datei `util.py` bietet Funktionen für das Lesen und Schreiben von Dateien sowie für das Aus-

lesen von GPU-Werten und die Erstellung von MD5-Hashwerten. Der Inhalt der Datei `main.py` ist ähnlich in Phasen strukturiert, wie es in Kapitel 5.1 vorgestellt wird. Sie verwendet die zuvor beschriebenen Dateien, um den gesamten Prozess bis zur Erstellung des Test-Tools auszuführen.

Ausgabeordner

Der Ausgabeordner enthält alle Dateien, die während des gesamten Prozesses generierten werden. Die vom Swagger Codegen generierten API-Client-Dateien werden hier übergeordnet abgespeichert. Alle anderen Dateien werden in einem Ordner abgelegt, dessen Name sich aus dem aktuellen Datum und der Uhrzeit zusammensetzt. In diesem Ordner befinden sich die verwendeten Prompts und die entsprechenden Python-Dateien, die mithilfe der LLM-Komponente erstellt werden. Außerdem sind hier die während des Prozesses generierten Logdateien und Tabellen sowie das Test-Tool-Programm zu finden, das sich im Ordner `generated-test-tool` befindet.

5.3 Test-Tool

Nachdem der Aufbau des Programms beschrieben wurde, welches das Test-Tool für die eigentlichen Tests konstruiert, soll eben diese Komponente nun im Detail betrachtet werden. Die Abbildung 10 zeigt hierbei die einzelnen Prozesse innerhalb der Komponente. Bevor das Test-Tool den Vorgang beginnen kann, muss

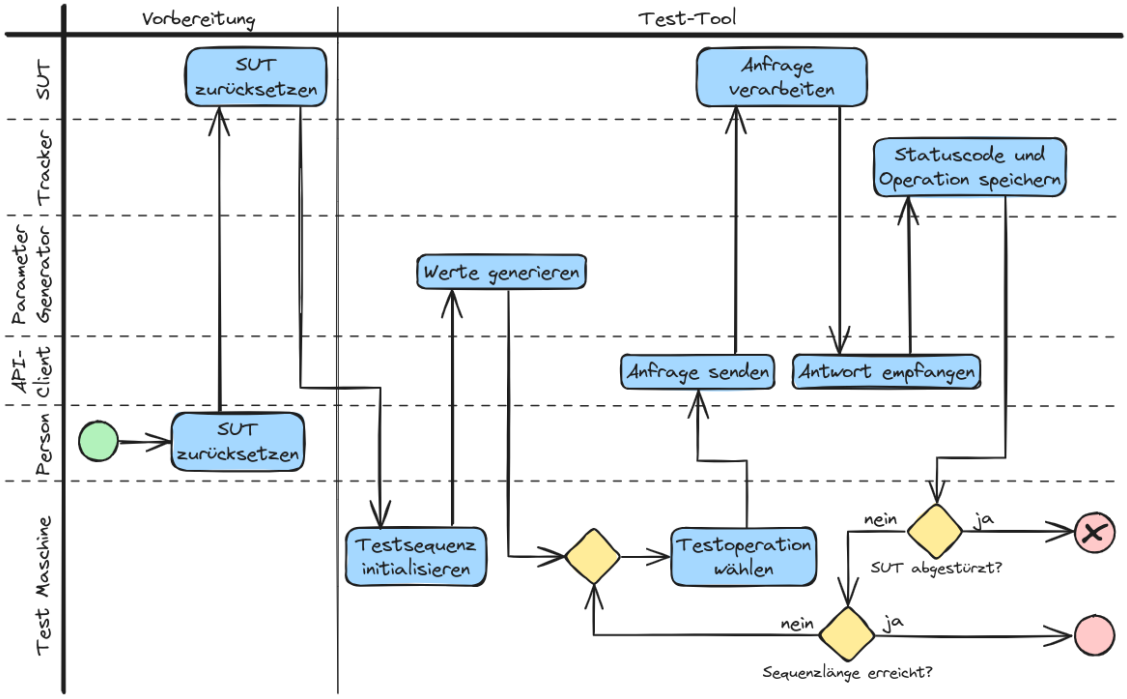


Abbildung 10: Prozesse des Test-Tools (vgl. [Kis21])

das SUT zurückgesetzt werden, um sicherzustellen, dass es bei jeder Testdurchführung den gleichen Zustand einnimmt. Das Zurücksetzen des SUTs erfolgt in diesem Fall durch eine Person. Es wäre jedoch möglich, diesen Prozess zu automatisieren, wie in der Arbeit von [Kis21] demonstriert wird. Sobald das SUT in seinen Ausgangszustand versetzt wurde, kann im nächsten Schritt das Test-Tool ausgeführt werden. Das Programm beginnt initial mit der Erstellung von Listen von Parameterwerten. Dabei wird für jeden Property bzw. Endpunkt, der über solche Eingabeparameter verfügt, eine separate Liste angefertigt. Hierfür werden die zuvor von der LLM-Komponente generierten Generatoren verwendet. Im nächsten Schritt werden zufallsbasiert die Operationen ausgewählt, mit denen das SUT getestet werden soll. In Verbindung mit den erstellten Parameterwerten und dem von Swagger Codegen generierten API-Client werden anschließend HTTP-Anfragen erstellt und an das SUT gesendet. Das SUT verarbeitet die Anfrage und sendet seine Antwort an den API-Client zurück. Ein Tracker protokolliert dabei die verwendete Operation mit den entsprechenden Parameterwerten und der dazugehörigen Antwort in einer separaten Datei. Anschließend wird überprüft, ob das SUT aufgrund eines Fehlers abgestürzt ist. Liegt ein solcher Fehler vor, wird der Prozess an dieser Stelle beendet. Andernfalls wird geprüft, ob die festgelegte Anzahl an Testfällen erreicht wurde. Falls dies der Fall ist, wird der Prozess ebenfalls beendet. Andernfalls wird die nächste Operation ausgewählt.

5.4 Web-API

Um die Tauglichkeit des generierten Test-Tools zu überprüfen, muss zunächst eine geeignete Web-API bereitgestellt werden, da diese für die späteren Tests benötigt wird. In diesem Kapitel wird daher kurz der Aufbau dieser API vorgestellt. Hierfür wird zunächst erst einmal das Datenbankschema, welches in Abbildung 11 dargestellt ist, betrachtet. Als Vorlage dient das GitHub-Projekt von [Aré22], das anschließend noch modifiziert wurde. Neben der Anpassung des Datenbankschemas wurde das Skript, das zufällige Einträge generiert, durch ein Skript ersetzt, das statische Einträge in der Datenbank erzeugt. Dies stellt sicher, dass der Datenbank-Container bei jedem Neustart denselben Zustand wiederherstellt. Anhand des dargestellten Schemas ist erkennbar, dass Informationen über Personen und deren Bestellungen gespeichert werden können, wodurch die folgenden Aktionen für eine Person ermöglicht werden:

Datenbank

- Eine Person kann seine eigenen Informationen einsehen.
- Eine Person kann seine eigenen Informationen ändern.

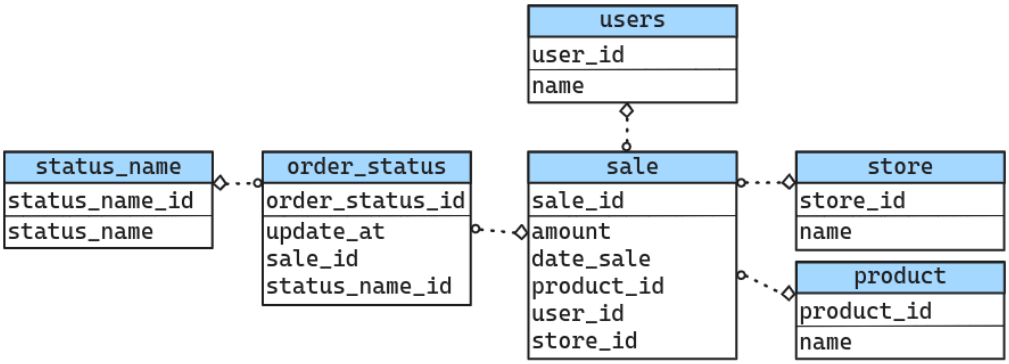


Abbildung 11: Datenbankschema (ER-Diagramm)

Methode	Pfad
GET	/user
PUT	/user
GET	/products
GET	/orders
POST	/orders
GET	/orders/{position}
PUT	/orders/{position}
DELETE	/orders/{position}

Tabelle 5: Operationen zur API

- Eine Person kann eine Liste aller verfügbaren Produkte abrufen.
- Eine Person kann ein Produkt aus dieser Liste bestellen und die gewünschte Menge angeben.
- Eine Person kann seine Bestellung stornieren.
- Eine Person kann seine Bestellung ändern.
- Eine Person kann seine getätigten Bestellungen auflisten.
- Eine Person kann Details zu einer bestimmten Bestellung einsehen.

Web-API

Die hier aufgezählten Aktionen lassen sich schließlich, wie in Tabelle 5 dargestellt, als Operationen für die Web-API zusammenfassen und können auch so implementiert werden.

6 Testaufbau und Durchführung

Der erste Abschnitt stellt die verschiedenen Systemumgebungen vor und beschreibt ihren Verwendungszweck. Im zweiten Abschnitt wird erläutert, welche Modelle für den Optimierungsprozess verwendet werden sollen. Anschließend wird erklärt, wie die Tests zur Optimierung der Modellausgaben aufgebaut und durchgeführt wurden. Das letzte Kapitel befasst sich mit der Evaluierung des Test-Tools und den damit verbundenen Schwierigkeiten.

6.1 Systemumgebungen

Systemumgebung 1

Die Durchführung erfolgte auf mehreren Systemumgebungen. Systemumgebung 1 verwendet als Betriebssystem die Ubuntu Version 22.04.2 LTS und ist ein 64-bit System mit 32 GB Arbeitspeicher. Als Prozessor ist eine AMD Ryzen 7 1700 CPU mit 8 Kernen und einer Taktfrequenz von 3 GHz eingebaut. Als GPU wird eine GeForce GTX 1060 mit 6GB RAM verwendet. Als Speicher für die persistenten Daten wird zudem eine Solid State Drive (SSD) mit einer Schreibgeschwindigkeit von 510 MB/s und einer Lesegeschwindigkeit von 555 MB/s verwendet. Bei

Systemumgebung 2

Systemumgebung 2 handelt es sich um einen gemieteten Google Colab Server. Als Abo-Modell wurde die Colab Pro+ Version erworben. Es stehen hierbei mehrere Laufzeitumgebungen zur Auswahl. Für alle Testdurchführungen wurde allerdings ausschließlich eine Laufzeitumgebung mit einer NVIDIA A100-SXM4-40GB GPU verwendet. Zudem werden in dieser Laufzeitumgebung etwa 83 GB an Arbeitspeicher zur Verfügung gestellt und etwa 200 GB an Festplattenspeicher. Als CPU wird eine Intel(R) Xeon(R) mit 6 Kernen und einer Taktfrequenz von 2,2 GHz bereitgestellt. Als Betriebssystem ist die Ubuntu Version 22.04.3 LTS installiert. Bei der Colab Pro+ Version werden jeden Monat 500 Recheneinheiten zur Verfügung gestellt. Die Laufzeitumgebung mit der A100 GPU verbraucht etwa 11,77 Recheneinheiten pro Stunde. Werden diese nicht verbraucht, dann verfallen diese nach 90 Tagen. Weil die Daten nach beendigung der Session automatisch gelöscht werden, habe ich zusätzlich noch 100 GB an Speicher über Google Drive erworben und mein Google Drive Konto mit den Google Colab Server verbunden. Das war aufgrund der teilweise sehr großen Modelle mit 40 bis 90 GB und den damit verbundenen langen Downloadzeiten notwendig.

Systemumgebung 3

Systemumgebung 3 ist ein Raspberry Pi 4 Model B Rev 1.2 mit 4 GB an Arbeitspeicher. Der Prozessor ist ein ARM Cortex-A72 mit 4 Kernen und einer Taktrate von 1,5 GHz. Als Betriebssystem ist ein Debian GNU/Linux der Version 12 installiert. Zusätzlich ist

auch noch eine Dockerumgebung mit der Version 24.0.7 eingerichtet. Als Speicherkarte wird eine microSD mit einer Schreibgeschwindikeit von 10 MB/s und einer Lesegeschwindigkeit von 98 MB/s verwendet.

Verwendung der Testumgebungen

In Systemumgebung 3 ist die zu testende Web-API mit der dazugehörigen Datenbank eingerichtet. Die Web-API wurde unter Verwendung des Node-Frameworks Express entwickelt und in einem Docker-Container bereitgestellt. Die dabei verwendete Node Version ist die 18.20.1. Als Datenbank wird ein PostgreSQL Container mit der Image-Version 15.2-alpine bereitgestellt. Damit sichergestellt wird, dass sich das SUT zu Beginn einer Testsequenz stets im selben Zustand befindet, wird vor der Ausführung immer ein neuer Container gestartet. Dabei werden auch alle persistenten Daten wieder zurückgesetzt. Neben der Web-API wird zusätzlich auch noch Swagger Codegen mit der Image-Version 3.0.54 als Docker-Container zur Verfügung gestellt. Systemumgebung 2 ist ausschließlich für die Tests bzw. für die Optimierungen der LLMs verwendet worden. Auf Systemumgebung 1 erfolgte die Ausführung des Test-Tools, welches die Web-API testet.

6.2 LLMs für die Durchführung der Tests auswählen

LLMs für Code

Um das Programm zum Testen der Web-API konstruiert zu können, müssen, neben dem API-Client, vorher noch die Funktionen bereitgestellt werden, die für die Erzeugung zufälliger Werte für die Eingabeparameter zuständig sind. Hierfür soll ein bereits vortrainiertes LLM verwendet werden. Bei der Suche nach geeigneten LLMs, die auf das Generieren von Code trainiert sind, konnten die folgenden Modelle identifiziert werden: Starcoder [Li+23] und Starcoder2 [Loz+24] von BigCode, DeepSeek-Coder [Guo+24] von DeepSeek-AI, Code Llama [Roz+23] von Meta und auch ein nicht-offizielles Code Llama-Repository. In Tabelle 6 werden diese Modelle nochmal zusammengefasst. Alle hier aufgelisteten Modelle werden auch in verschiedenen Modellgrößen angeboten. Die Checkpoints, mit einer Modellgröße von fast 70B Parametern, können nicht vollständig von der Grafikkarte geladen werden. Um diese Modelle nutzen zu können, wäre es erforderlich, einige Layer mithilfe der Device-Map auf verschiedene Gerätespeicher zu verteilen. Jedoch wird aufgrund der daraus resultierenden langen Inferenzzeiten von einer solchen Umsetzung abgesehen. Checkpoints mit einer Modellgröße von 33B oder 34B Parametern können verwendet werden, sofern ihre Präzision auf 8 oder 4 Bit reduziert wird. Die Quantisierung von Modellen führt ebenfalls zu längeren Inferenzzeiten, diese sind jedoch im Vergleich deutlich kürzer als wenn man die Modell-Layer auf verschiedenen Gerätespeichern verteilt. Basierend auf

Modell	Architektur	Größe in Mrd.	Quelloffen?
Starcoder	GPTBigCodeForCausalLM	1, 3, 7 & 15.5	ja
Starcoder2	Starcoder2ForCausalLM	3, 7 & 15	ja
Code Llama für Python	LlamaForCausalLM	7, 13, 34 & 70	eingeschränkt
Code Llama* für Python	LlamaForCausalLM	7, 13, 34 & 70	ja
DeepSeek-Coder	LlamaForCausalLM	1.3, 7 & 33	ja

* nicht offizielles Code Llama Repository

Tabelle 6: Modelle

den Informationen aus den Papern verwenden alle Modelle zudem eine Architektur, die ausschließlich aus einem Decoder besteht. Weiterhin handelt es sich bei beinahe allen Modellen um Open Source Projekte. Eine Ausnahme bildet hierbei das Model von Meta, welches nur eingeschränkt verwendet werden darf.

Starcoder Dataset

Zunächst werden erst einmal die Datensets betrachtet, mit denen die verschiedenen Modelle trainiert wurden. Das Starcoder-Modell wurde mit dem Datenset „The Stack v1.2“ [Koc+22] trainiert. Dabei hat man die dort enthaltenen Daten vor Beginn des Trainings mit einer Kombination aus heuristischen Filtern und manuellen Überprüfungen bereinigt, wodurch am Ende in etwa 800 GB an Trainingsdaten verwendet werden konnten. Der dort enthaltene Anteil an Python-Code beträgt dabei etwa 60 GB. StarCoderBase wird mit etwa 1 Trill. Tokens aus diesem Datenset trainiert. Ein anderes, auf Python spezialisiertes Modell, wird zusätzlich noch mit 35 Mrd. an Python Tokens trainiert. Das Starcoder2-Modell hat man wiederum mit dem Datenset „The Stack v2“ trainiert. Ähnlich wie bei der Vorgänger-Version hat man die dort enthaltenen Daten wieder dedupliziert und bereinigt. Weil der Anteil einiger Programmiersprachen deutlich überwiegte, hat man zusätzlich noch ein „Downsampling“ durchgeführt. Dabei wurde der Anteil einer Programmiersprache auf maximal 200 GB begrenzt. Zudem hat man auch hier wieder zwei verschiedene Datensets konstruiert, mit denen die Modelle trainiert wurden. Der Checkpoint mit den 15 Mrd. Parametern wurde mit allen 619 enthaltenen Programmiersprachen trainiert, während die beiden kleineren Checkpoints mit einem kleineren Datenset trainiert wurden, dass nur 17 verschiedene Programmiersprache enthält. Bei dem kleineren Datenset ist die Programmiersprache Python ebenfalls vorhanden. Der Anteil an Python-Code im Datensatz beträgt etwa 190 GB, was dreimal mehr ist als beim Vorgänger. Neben den Programmiersprachen werden auch Sprachen, die zur Dokumentation von Code verwendet werden, beim Training verwendet. Die kleineren Modelle werden

Modell	Checkpoint	Präzision
Starcoder2	bigcode/starcoder2-7b	bfloat16
	bigcode/starcoder2-15b	bfloat16
DeepSeek Coder	deepseek-ai/deepseek-coder-6.7b-base	bfloat16
	deepseek-ai/deepseek-coder-33b-base	4bit
Code Llama für Python	codellama/CodeLlama-7b-Python-hf	bfloat16
	codellama/CodeLlama-34b-Python-hf	4bit (Contrastive) 8bit (Sonstiges)

Tabelle 7: Die beim Fine-Tuning betrachteten Modelle

mit etwa 622 Mrd. bis 658 Mrd. Tokens und das größere mit etwa 913 Mrd. Tokens trainiert. Die Code Llama Modelle wurden ausgehend von den Llama 2 Modellen trainiert. Dabei verwendete man einen neuen Datensatz, welcher grötenteils aus öffentlich zugänglichen Code (859 GB - 85%) besteht. Beim Training der Code Llama Modelle wurden insgesamt 500 Mrd. Tokens aus dem Datensetzt verwendet und Code Llama für Python wurde zusätzlich noch mit weiteren 100 Mrd. Tokens aus einem anderen Datenset trainiert. Dieses Datenset umfasst etwa 79 GB an Python Code. Auch hier hat man die Datensets weitesgehend dedupliziert und soweit gefiltert, bis ausschließlich Code mit „Textbuch“-Qualität [Gun+23] übrig blieb. Das Datenset, welches für das Training von DeepSeek Coder verwendet wurde, besteht ebenfalls zu 87% aus Open-Source Code und 10% aus codebezogenen natürlichen Texten. Um die Qualität des Datensets zu verbessern, hat man dieses wieder dedupliziert und verschiedene Filterregeln angewendet. Nach der Bereinigung betrag der Python-Anteil im Datensatz etwa 120 GB. DeepSeek Coder wird in zwei verschiedenen Varianten angeboten: DeepSeek-Coder-Base und DeepSeek-Coder-Instruct. Beide Modelle wurden von Grund auf mit 2 Trill. Tokens trainiert.

Code Llama Dataset

DeepSeek Coder Dataset

Für die Überprüfung der Modelle hat man sich schließlich entschieden drei Modelle mit jeweils zwei Checkpoints zu betrachten. Das offizielle Code Llama Modell habe ich aufgrund der Einschränkungen ausgeschlossen. Alternativ soll hierfür das nicht offizielle Modell in die Untersuchungen mit einbezogen werden. Weiterhin wird auch von den BigCode-Modellen auschließlich Starcoder2 für die weiteren Untersuchungen betrachtet. Das Modell konnte nicht nur eine bessere Leistung als seine Vorgängerversion erzielen [Loz+24], sondern verfügte auch über eine angepasste Architektur, die sich sowohl bei den Modellen von Meta als auch bei denen von DeepSeek-AI als bewährt bewies. In Tabelle 7 werden die für die Durchführung verwendeten Checkpoints zusammengefasst.

Auswahl der Modelle

6.3 Optimierung der Prompts

Damit das Test-Tool sinnvolle Werte für die Eingabeparameter der verschiedenen Endpunkte generieren kann, muss die LLM-Komponente die dafür notwendigen Generatoren bereitstellen. Dazu ist es erforderlich, geeignete Anweisungen zu konstruieren und an die LLM-Komponente zu übergeben. Dieser Vorgang sollte mit minimalem Aufwand ermöglicht werden. Daher wird eine wiederverwendbare Prompt-Vorlage erstellt und mit der OAS kombiniert, um die verschiedenen Generatoren möglichst automatisiert zu erzeugen. Ziel des Optimierungsprozesses ist es, eine Vorlage zu entwickeln, die diese Anforderungen erfüllt. Hierfür wurden beim Optimierungsprozess die folgenden Komponenten betrachtet und angepasst: Prompt-Vorlagen, OAS und Modellparameter. Es wurden zusätzlich auch noch verschiedene LLMs untersucht, um zu überprüfen, ob durch die Verwendung desselben Prompts mit einem anderen Modell ein besseres Ergebnis erzielt werden kann.

Dekodierstrategien

Für die Tests zur Optimierung der Modellausgaben werden, wie in Kapitel 6.2 beschrieben, insgesamt drei Modelle mit je zwei Checkpoints untersucht. Für jeden Checkpoint wurden jeweils drei verschiedene, weitgehend identische Konfigurationsdateien erstellt. Diese Konfigurationsdateien unterscheiden sich lediglich in den verwendeten Modellparametern, da insgesamt drei unterschiedliche Dekodierstrategien bei der Untersuchung betrachtet werden sollen. Es werden mehrere unterschiedliche Verfahren gewählt, um die Ergebnisse miteinander zu vergleichen und zu untersuchen, ob eine Strategie generell bessere Resultate liefert als die anderen. Beam Search und darauf basierende Verfahren wurden aufgrund der damit verbundenen Generierungszeiten ausgeschlossen. Wie bereits in Kapitel 3.2 erläutert, generiert Beam Search zunächst mehrere unterschiedliche Antworten, was zu einer drastischen Erhöhung der Generierungszeit führt, abhängig von der Anzahl der möglichen Kandidaten. Für die Experimente wurden daher Greedy Search, Contrastive Search und ein Sampling-Verfahren ausgewählt. Für das Sampling-Verfahren wurde explizit Nucleus Sampling ausgewählt. Die in den Verfahren Nucleus Sampling und Contrastive Search verwendeten Parameterwerte sind in Tabelle 8 aufgeführt.

Vorgehensweise

Bevor die Tests durchgeführt werden konnten, musste zunächst eine Prompt-Vorlage in Kombination mit einer OAS erstellt werden. Da diese Vorlage mit der OAS während der verschiedenen Testphasen kontinuierlich modifiziert werden, erfolgt eine Versionierung, wie in Kapitel 3.2 beschrieben, jedoch ohne den Einsatz eines speziellen Tools. Dies ermöglicht eine eindeutige Zuordnung der Tests zu den verwendeten Prompt- und OAS-Versionen. Alle Tests, die mit derselben

Parameter	Wert
do-sample	True
temperature	Default: 1.0
top-p	0.95

(a) Nucleaus Sampling

Parameter	Wert
top-k	5
penalty_alpha	0.6

(b) Contrastive Search

Tabelle 8: Bei Nucleaus Sampling (links) und Contrastive Search (rechts) verwendete Hyperparameter

Prompt- und OAS-Version durchgeführt wurden, werden dabei zu einer gemeinsamen Testgruppe zusammengefasst. Innerhalb dieser Testgruppen wurde zudem noch zwischen den Tests differenziert, die mit unterschiedlichen Modellen durchgeführt wurden.

Auswertung & Modifikationen

Nach Abschluss aller Tests einer Testgruppe wurden die Ergebnisse ausgewertet. Anhand der gewonnenen Daten sollten mögliche Probleme identifiziert und Strategien zur Verbesserung entwickelt werden. Nach den als notwendig erachteten Anpassungen konnten die Tests für die nächste Gruppe durchgeführt werden. Dieser Vorgang wurde mehrmals wiederholt, mit dem Ziel die Ausgabe des LLMs immer weiter zu optimieren. Darüber hinaus werden auch Informationen über das System, wie z. B. Details über die verwendete CPU, GPU usw., auf dem die Tests durchgeführt wurden, ebenfalls dokumentiert. Damit die Ergebnisse zudem auch möglichst einfach und mit geringen Aufwand in dem dafür vorgesehenen Dokument integriert werden können, werden die notwendigen Tabellen, soweit dies möglich ist, automatisch generiert und ausgefüllt.

Sonstiges

Aufgrund des begrenzten Cloudspeichers ist es mir nicht möglich alle Checkpoint auf dem Speicher zu hinterlegen. Um sämtliche Checkpoints einer Testgruppe evaluieren zu können, müssen Modelle gelöscht und erneut heruntergeladen werden, was aufgrund der Größe einiger Checkpoints zeitaufwändig ist. Daher werden zunächst nur zwei Checkpoints desselben Modells getestet. Anfangs hat man daher nur die Checkpoints vom Starcoder2-Modell untersucht.

6.4 Evaluierung des Test-Tools

Schwierigkeiten

Eine umfassende Bewertung des Test-Tools gestaltet sich als problematisch. Der in Kapitel 6.3 beschriebene Optimierungsprozess hat sich als äußerst zeitaufwändig erwiesen, um einen geeigneten Prompt für ein Modell zu konstruieren, der die Generierung der notwendigen Generatoren für das Test-Tool ermöglicht. Gleichzeitig konnten aufgrund des Zeitmangels essenzielle Prozesse wie die automati-

sche Überprüfung mit einem Orakel wie der OAS nicht mehr implementiert werden. Derzeit wird ausschließlich überprüft, ob das SUT abstürzt, was die Analyse der Fehlerursache erschwert. Aus diesem Grund konzentriert sich die Bewertung darauf, ob das Test-Tool grundlegende Funktionen erfüllt, wie beispielsweise das Erstellen von Anfragen an die Web-API und das Empfangen von Antworten. Dennoch wird im Kapitel 8 eine detaillierte Betrachtung und ein Vergleich mit anderen bestehenden Lösungen durchgeführt.

Erstellen der OAS

Um das SUT mit einem verhältnismäßigen Aufwand testen zu können, hat man für die Versuchsdurchführungen festgelegt, dass bei einer Ressourcenabfrage keine IDs verwendet werden sollen, sondern stattdessen die Position der Ressource in der Datenbank abgefragt wird. Damit gezielte Anfragen zu den Ressourcen der Datenbank generiert werden können, müsste das Test-Tool den Zustand der Datenbank kennen. Weiterhin wird festgelegt, dass alle Aktionen ausschließlich von einer einzelnen Person getätigt werden dürfen. Ansonsten gelten alle Einschränkungen, die in Kapitel 4.2 genannt wurden. Basierend auf diesen Anforderungen wird eine OAS erstellt, die zu den in Kapitel 5.4 beschriebenen Operationen passt.

7 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst. Zunächst werden die Resultate analysiert, die während der Optimierung der Modellausgaben dokumentiert wurden. Anschließend wird das Kapitel mit einer kurzen Betrachtung des Test-Tools abgeschlossen.

7.1 Optimierung der Modellausgabe

Bevor die eigentlichen Ergebnisse betrachtet werden, sollen zunächst die Hauptprobleme und die daraus resultierenden Modifikationen der verschiedenen Gruppen zusammengefasst werden. Dies soll einen konkreten Überblick über die unterschiedlichen Gruppen vermitteln, um ein besseres Verständnis für die Betrachtung der Ergebnisse zu ermöglichen. Im weiteren Verlauf werden die Gruppen durch entsprechende Kürzel bezeichnet. Eine Gruppe mit der Bezeichnung V4V5 bedeutet beispielsweise, dass alle Tests mit der Prompt-Vorlage der Version 4 und der OAS-Version 5 durchgeführt wurden.

Gruppe V1V1

Die erste Prompt-Vorlage beginnt mit einer Rollenzuweisung, die das LLM dazu veranlassen soll, als „Codegenerator“ zu agieren. Daraufhin folgt ein Beispiel, dass eine Aufgabenbeschreibung, eine Gruppe von Parametern und deren Eigenschaften, eine CoT-Anweisung und eine Musterlösung umfasst. Die Musterlösungen werden dabei stets durch dreifache Anführungszeichen separiert. Im nächsten Abschnitt werden die Parameter aus der verwendeten OAS hinterlegt, wobei dieser Teil ähnlich strukturiert ist wie das Beispiel im vorherigen Abschnitt. Als Auszeichnungssprache hat man Markdown verwendet. Die OAS für die erste Testgruppe enthält insgesamt vier Endpunkte und umfasst verschiedene Datentypen wie Boolean, Integer, Number-Datentypen mit verschiedenen Formaten sowie Strings mit verschiedenen Formaten und Patterns.

Probleme von V1V1 & Modifikationen für V2V2

Wie in Listing 6 dargestellt, hat die LLM-Komponente nach dem Codeblock zusätzliche Tokens generiert. Meistens solange bis die maximale Anzahl an möglichen Tokens erreicht war. Aufgrund der Tatsache, dass die Ausführungszeit direkt proportional zur Anzahl der generierten Tokens ist, führte dies ebenfalls zu erheblich längeren Generierungszeiten (siehe Anlage G: Diagramme). In allen erzeugten Dateien wurde der Codeblock jedoch immer mit dreifachen Anführungszeichen (Zeile 11) beendet. Um dieses Problem zu beseitigen, wurde

```
import random

def get_dict_with_random_values() -> dict:
  variables = {
    "test1": random.randint(1,100),
    "test2": random.getrandbits(32),
    "test3": random.getrandbits(64),
    "test4": random.choice([True, False])
  }
  return variables
```
<file_sep>/src/01_variables.py
Variables are used to store information [...]

Listing 6: Ausschnitt aus 2024-05-14 12:51:41 - snip-p01-b01.py

daher für die zweite Testgruppe diese Zeichensequenz als EOS-String zur Liste hinzugefügt. Häufig wurden bei den numerischen Datentypen Bereiche zwischen 1 und 100 festgelegt, möglicherweise aufgrund der Verwendung dieser Grenze im Beispiel des Prompts. Manchmal wurden auch Grenzen für unsigned Integer erstellt, wenn das Format spezifiziert wurde, wie `int32` oder `int64`. Die Swagger-Dokumentation schreibt jedoch vor, dass es sich um signed Integer handeln muss, weshalb dies als Fehler betrachtet wird. Auch bei den Strings fällt wieder auf, dass, wenn nichts anderes spezifiziert ist, die generierten Ergebnisse dem Beispiel aus dem Prompt sehr ähneln. Abgesehen davon werden teilweise auch statische Werte zugewiesen und die definierten String-Patterns werden nicht immer korrekt angewendet. Zur Bewältigung der Probleme mit numerischen und String-Datentypen wurde die Verwendung von Formatspezifikationen oder Schlüsselwörtern wie `minimum` oder `maximum` nun als notwendig betrachtet und die OAS entsprechend angepasst. Man hat dies entsprechen auch so im Bespiel des Prompts übernommen. Weiterhin wurde auch noch die CoT-Anweisung und die Musterlösung im Beispiel angepasst. Bei der CoT-Anweisung hat man die Liste an Anweisungen gelöscht. Für jeden Eingabeparameter wird zudem eine eigene Funktion bereitgestellt, die nur dafür zuständig ist, einen zufälligen Wert für diesen Parameter zu erzeugen. Die letzte Funktion verfügt über das Dictionary und den vom Tester zugewiesenen Funktionsnamen. Dabei werden die verschiedenen Zwischenschritte mit Kommentaren erläutert, um die CoT-Vorgehensweise auch nochmal in der Musterlösung deutlicher zu machen. Ein weiteres Problem besteht darin, dass teilweise auch Packages verwendet werden, die nicht importiert werden. Das Hauptprogramm wurde daher durch eine Funktion ergänzt, die auf solche fehlenden Imports überprüft (siehe Kapitel 5.1).

Probleme von V2V2 & Modifikationen für V3V3

Die Ergebnisse der zweiten Testgruppe zeigen, dass die Formate `password`, `byte` und `binary` weiterhin problematisch sind. Die verfügbaren Informationen scheinen nicht auszureichen, damit der Generator angemessene Lösungen erzeugen kann. Daher werden diese Formate aus der OAS für die Testgruppe entfernt. Des Weiteren wurde beobachtet, dass die generierten Python-Dateien diverse Fehler enthalten, sowohl Syntax- als auch Semantikfehler. Zudem werden teilweise zu umfangreiche Programme erzeugt, welche die maximale Anzahl an Tokens überschreiten, was zu unvollständigen Programmen führt. In einigen Fällen weichen die Programme während des Lösungsversuchs ab und generieren zusammenhangslosen Text. Da die Ursache für diese Fehler nicht eindeutig bestimmt werden konnte, wurde die Prompt-Vorlage um ein weiteres Beispiel ergänzt. Es wurden teilweise auch leere Dateien erzeugt. Daher wurde für die nächste Testgruppe eine Resampling-Strategie implementiert (siehe Kapitel 5.1).

Probleme von V3V3 & Modifikationen für V4V4

In der Testgruppe V3V3 treten weiterhin viele Fehler bei den String-Datentypen im Format `date` oder `date-time` auf. Beispielsweise werden Monate im Bereich von 0 bis 99 generiert oder es werden nur Werte von 1970 bis 1973 erzeugt. Häufig wird auch ein Datum, das als `date` definiert ist, als `date-time` ausgegeben. Zudem werden weiterhin Lösungen für Strings mit definierten Mustern häufig falsch umgesetzt. Erst nach Abschluss der Tests der Testgruppe V3V3 wurde auch noch ein Fehler bemerkt, der sowohl in der Prompt-Vorlage als auch in der OAS vorhanden ist. Dadurch war auch die in der Prompt-Vorlage vorgegebene Musterlösung fehlerhaft. Um diesen Fehler zu beheben, wurde das Pattern `[.]{1,20}` in `.{1,20}` geändert. Da in Python RegEx-Ausdrücke stets mit einem `r` gekennzeichnet werden, wird dies in der Testgruppe V4V4 auch in der Prompt-Vorlage berücksichtigt. Für die nächste Testgruppe werden nun auch alternative Modelle eingesetzt, um zu überprüfen, ob diese bessere Ergebnisse erzielen können als das zuvor verwendete Starcoder2-Modell.

Probleme von V4V4 & Modifikationen für V4V5

Bei den Dateien, die durch das DeepSeeker-Modell generiert wurden, sind auch einige mit dem Token `<|end_of_sentence|>` beendet worden. Für die Testgruppe V4V5 wird in den Konfigurationsdateien für das DeepSeeker-Modell daher noch der EOS-Token `<|end_of_sentence|>` ergänzend hinzugefügt. Auch in der Testgruppe V4V4 tritt das gleiche Problem mit den Strings auf, wie es bereits in der Testgruppe V3V3 beobachtet wurde. Keines der verwendeten Modelle konnte für die String-Datentypen zuverlässig alle vier Kriterien erfüllen. Allerdings bedeutet dies nicht, dass alle Funktionen für die String-Datentypen fehlerhaft sind, sondern dass mindestens eine Funktion innerhalb des generierten Programms einen Fehler enthält. Dadurch das jedes Property über mehrere Variablen verfügt, kommt es nur äußerst selten vor, dass auch alle spezifizierten Variablen korrekt sind.

Es werden also teilweise auch Funktionen erstellt, die eine korrekte Teillösung darstellen. Daher werden an der Prompt-Vorlage diesmal keine Änderungen vorgenommen. Stattdessen wird die OAS so modifiziert, dass jedes Property genau einen Eingabeparameter besitzt. Ziel ist es, zu ermitteln, bei welchen Datentypen und zugewiesenen Eigenschaften die verschiedenen LLMs die besten Lösungen generieren können.

> Anmerkung: Nach Abschluss der Tests wurde ein weiterer Fehler identifiziert, der bereits seit der Testgruppe V2V2 besteht und sich in der Musterlösung der Prompt-Vorlage befindet. Die Zeile mit dem Funktionsaufruf «`random.randint(3, 20)`» sollte zu «`random.randint(2, 50)`» geändert werden.

Vergleich der Gruppen

Nachdem alle Tests für die Gruppen durchgeführt und die Erfüllung der in Kapitel 4.4 definierten Kriterien überprüft wurden, hat man sogenannte Scores zusammengestellt, um die Qualität der Modellausgaben vergleichen zu können. Da sich die maximale Anzahl von Score-Punkte teilweise zwischen den Testgruppen unterscheidet, wurden die Werte zusätzlich noch normalisiert, um einen Vergleich der Qualität zwischen Testsequenzen aus unterschiedlichen Gruppen zu ermöglichen.

Zunächst wird das Säulendiagramm in Abbildung 12 betrachtet. Der Score-Wert ergibt sich aus der Summe aller Kriterien, die ein Modell in einer bestimmten Gruppe erfüllt hat. Es werden auch vorläufig die Testsequenzen der verschiedenen Dekodierstrategien zusammengefasst betrachtet. Damit soll zunächst erst einma ein Überblick darüber geschaffen werden, welche Testgruppe bzw. welches Modell die besten Ergebnisse auf Basis der genannten Modifikationen erzielt. Besonders bei den ersten drei Testgruppen (V1V1, V2V2, V3V3) konnte die Qualität der Modellausgaben durch die genannten Modifikationen signifikant verbessert werden. Während in der ersten Testgruppe lediglich knapp 50% der Kriterien erfüllt wurden, erreichte die Testgruppe V3V3 eine Erfüllungsrate von etwa 83%. Das Hinzufügen des EOS-Strings hatte vermutlich den größten Einfluss, da dadurch ein Großteil der generierten Programme ausführbar wurde. Das Ergebnis des Starcoder2-Modells in der Testgruppe V4V4 zeigt im Vergleich zur vorherigen Gruppe weder eine signifikante Verbesserung noch Verschlechterung. Trotz der vorgenommenen Modifikationen blieb die Qualität der Modellausgaben unverändert. Allerdings wurden leichte Unterschiede zwischen den Modellen innerhalb der Testgruppe V4V4 festgestellt. Insbesondere konnte mit der inoffiziellen Version von Code Llama für Python ein Ergebnis erzielt werden, das fast 5%

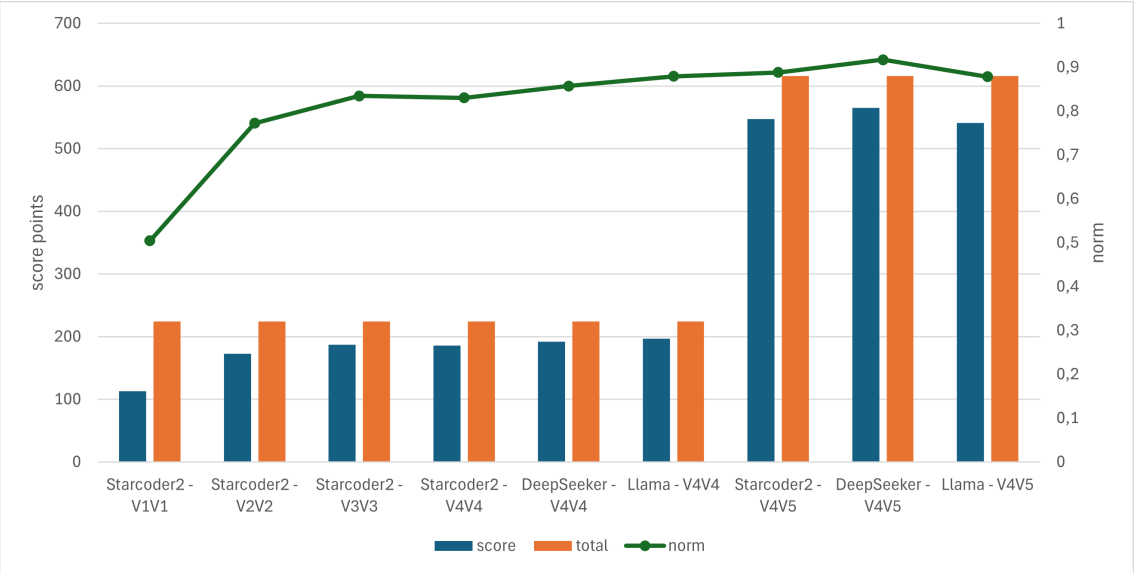
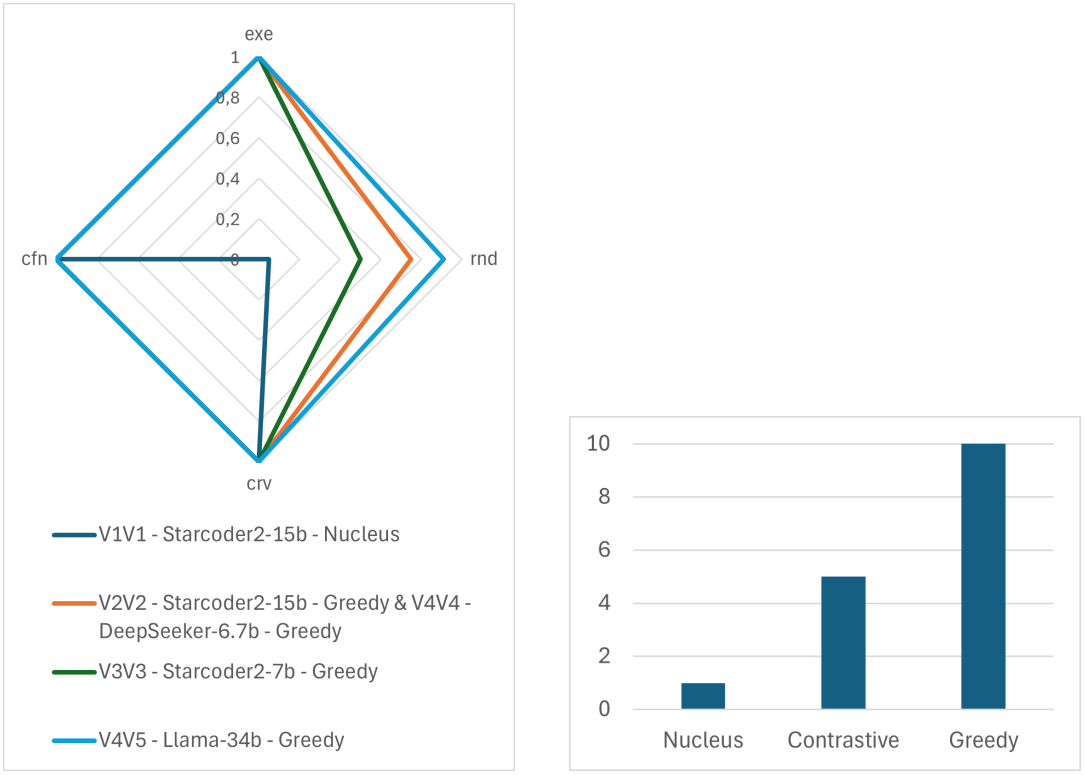


Abbildung 12: Säulendiagramm mit normalisierten Datenpunkten - Vergleich der Qualität der Modelle und der Testgruppen

besser war als das des Starcoder2-Modells. Darüber hinaus ist hervorzuheben, dass das DeepSeeker-Modell aufgrund eines fehlenden EOS-Strings teilweise nicht ausführbare Lösungen generierte. Dieser Mangel wurde erst in der nachfolgenden Testgruppe behoben, wodurch dann das DeepSeeker-Modell das beste Ergebnis erzielen konnte. Wie bereits erwähnt, wird in der Testgruppe V4V5 jedem Property in der OAS ein eigener Eingabeparameter zugeordnet, um die problematischsten Datentypen zu identifizieren. Dies führt zu einer Zunahme an definierten Properties und somit zu einer größeren Anzahl an Generatoren, die anschließend auf die Erfüllung der Kriterien überprüft werden. Wie in der Grafik zu sehen ist, weisen die Modelle in dieser Gruppe daher einen deutlich höheren Score-Wert auf als in den anderen Gruppen. In dieser Gruppe konnten erneut leicht bessere Ergebnisse als in der vorherigen Gruppe erzielt werden. Diesmal weist sogar das DeepSeeker-Modell das beste Ergebnis mit einer Erfüllungsrate von etwa 92% auf. Dieser Wert unterscheidet sich jedoch nicht signifikant von den Ergebnissen der anderen beiden Modelle. Die Datenreihe der normierten Score-Werte zeigt einen stetig zunehmenden Trend, der jedoch bereits in den ersten drei Testgruppen deutlich abflacht. Während die Qualität mit jeder Testgruppe steigt, ist dieser Anstieg ab der Testgruppe V3V3 nur noch geringfügig.

Eine weitere Erkenntnis ist, dass das Starcoder2-Modell in den Gruppen V4V4 und V4V5 nur geringfügig schlechter abschneidet als die beiden anderen Modelle, obwohl dessen größter Checkpoint nicht einmal halb so groß ist wie die größeren Checkpoints der anderen beiden Modelle.



(a) Radardiagramm (b) Säulendiagramm

Abbildung 13: Radardiagramm (links) - Vergleich der besten Modelle jeder Gruppe | Säulendiagramm (rechts) - Darstellung der Häufigkeit, mit der eine Dekodierstrategie in den verschiedenen Gruppen die beste Lösung generierte

Betrachtung der Kriterien & Dekodierverfahren

Während der Testdurchführungen wurden auch verschiedene Checkpoints mit unterschiedlichen Dekodierverfahren betrachtet. Für die weitere Analyse wurde für jede einzelne Testsequenz ein Score für jedes Kriterium erstellt. Um eine bessere Vergleichbarkeit während der Auswertung sicherzustellen, wurde jeweils das beste Modell mit der besten Strategie pro Gruppe ausgewählt und in einem Radardiagramm, welches in der Abbildung 13 dargestellt ist, abgebildet. Aus denselben Gründen wie zuvor wurden auch hier die Score-Werte wieder normalisiert. Ein jeder Eckpunkt des Radardiagramms repräsentiert hierbei einen Score für jedes Kriterium. Mit Ausnahme des Ergebnisses der Testgruppe V1V1 konnten alle anderen dargestellten Lösungsvarianten die Kriterien `exe`, `cfn` und `crv` stets vollständig erfüllen. Die erste Testgruppe konnte die Bedingung zur Ausführbarkeit nicht erfüllen, da der EOS-String fehlte, wodurch der Generierungsprozess nicht an der richtigen Stelle beendet werden konnte. Durch diese Darstellung wird nochmal deutlich, dass die Schwierigkeit darin besteht, eine geeignete Lösung zu finden, welche das Kriterium der Zufälligkeit erfüllt. Das Llama-Model aus der Testgruppe V4V5 konnte mit Greedy Search sogar ein nahezu fehlerfreies Ergebnis erzeugen. Während der Konstruktion des Radardiagramms fiel zudem auf,

dass das Dekodierverfahren Greedy Search in den verschiedenen Testgruppen meist die besseren Ergebnisse lieferte. Nucleus Sampling konnte dies nur einmal in der ersten Testgruppe erreichen. Das Säulendiagramm in Abbildung 13 zeigt, wie häufig eine bestimmte Strategie das beste Ergebnis erzielte. Hierbei wurde für jedes Modell pro Gruppe stets das Verfahren mit dem höchsten Score ausgewählt. Es kam auch häufig vor, dass verschiedene Strategien denselben Score erreichten. Da Nucleus Sampling einmal ein besseres Ergebnis als die anderen Strategien erzielen konnte, erscheint es als denkbar, dass man mit einer geeigneten Resampling-Strategie, solange neue Samples erzeugen könnte, bis eine Lösung gefunden wird, die alle Kriterien erfüllt. Auch wenn man mit diesem Verfahren Lösungen erzeugen könnte, die andere Dekodierstrategien nicht generieren können, würde dies jedoch insgesamt zu längeren Generierungszeiten führen.

Betrachtung der Datentypen

Zum Schluss werden noch die verschiedenen Datentypen getrennt voneinander betrachtet. Hierfür werden die Radardiagramme aus Abbildung 14 betrachtet. Die Ergebnisse wurden für jedes Modell aus der Testgruppe V4V5 nach den Datentypen kategorisiert und anschließend Scores für die verschiedenen Kriterien ermittelt. Grundsätzlich ist zu beobachten, dass alle Kriterien bis auf `rnd` nahezu

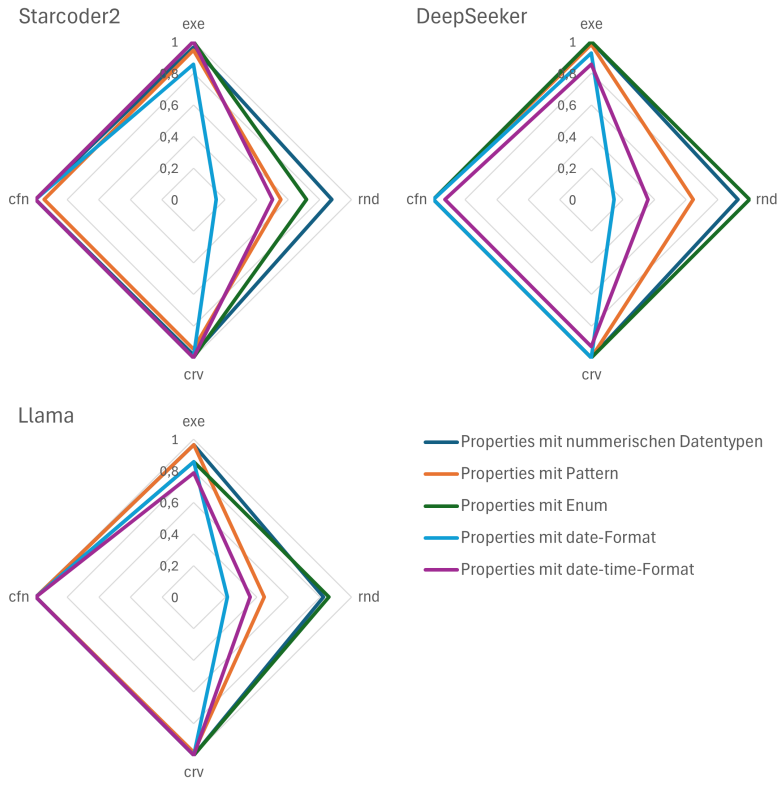


Abbildung 14: Erreichter Score bei den verschiedenen Property-Kategorien pro Modell

immer erfüllt werden. Zudem zeigen die Ergebnisse, dass Lösungen für numerische Datentypen und Datentypen mit einer Enumeration am besten abschneiden. Hierbei werden am häufigsten Lösungen generiert, die auch das Kriterium der Zufälligkeit erfüllen. Strings mit einem definierten Pattern liegen im mittleren Bereich, während Strings im `date`- oder `date-time`-Format den LLMs die größten Schwierigkeiten bereiten.

7.2 Überprüfung der Funktionalität des Test-Tools

Aufgrund der verbleibenden Zeit konnten einige notwendige Funktionalitäten für das Test-Tool nicht mehr implementiert werden, wodurch bestimmte wichtige Aspekte nicht untersucht werden konnten. Dennoch soll an dieser Stelle eine kurze Zusammenfassung dessen erfolgen, was mit dem Test-Tool erreicht wurde und welche Beobachtungen dabei gemacht wurden.

Gruppe V5V6

Nachdem die Optimierung der Modellausgabe abgeschlossen war, wurde nachträglich noch der Fehler in der Prompt-Vorlage korrigiert und die OAS passend zur Web-API konstruiert. Die in diesem Abschnitt betrachtete Testgruppe V5V6 umfasst somit auch das Test-Tool. Für dessen Konstruktion wurde im ersten Versuch der Checkpoint `codellama/CodeLlama-34b-Python-hf` verwendet. Aufgrund eines Fehlers musste die Funktion zur Konstruktion des Test-Tools jedoch angepasst und der Vorgang wiederholt werden. Im zweiten Versuch wurde der Checkpoint `bigcode/starcoder2-3b` eingesetzt. Dabei hat man bei beiden Ansätzen Greedy Search als Dekodierstrategie gewählt. Ursprünglich sollten die erzeugten Generatoren für die Parameterwerte mit denen des ersten Versuchs ausgetauscht werden. Obwohl das Starcoder2-Modell nur 3 Mrd. Parameter umfasst und somit deutlich kleiner ist, erzielte es dennoch ein leicht besseres Ergebnis. Daher wurde für die weitere Betrachtung die Lösung mit dem Starcoder2-Modell gewählt. Beide Ansätze hatten bei der selben Operationen mit String-Datentypen einen Fehler. Alle anderen Operationen konnten korrekt umgesetzt werden.

Betrachtung der Funktionalitäten

Die in Kapitel 5.3 beschriebenen Prozesse konnten größtenteils eingehalten werden. Lediglich der „Tracker“, der die ausgeführten Operationen und die zugehörigen Statuscodes chronologisch speichern sollte, ist im verwendeten Test-Tool nicht vorhanden. Stattdessen wurde die Debug-Konsole genutzt, um die Korrektheit des Test-Tools bzw. der Web-API zu überprüfen. Da ein automatisiertes Verfahren zur Überprüfung der Ergebnisse unter Verwendung eines Orakels fehlte, konnte dies nur eingeschränkt untersucht werden. Die Ergebnisse haben aber grundsätzlich gezeigt, dass alle in der OAS spezifizierten Operationen im

Programm vorhanden sind und unter Verwendung der erstellten Client-API ausgeführt werden konnten, ohne das noch Anpassungen vorgenommen werden mussten. Das SUT, das die Anfragen des Test-Tools erhielt, konnte diese auch erfolgreich verarbeiten und eine entsprechende Antwort zurücksenden, welche anschließend über die Konsole dargestellt wurde. Diese Ergebnisse wurden parallel auch noch mit der Datenbank abgeglichen.

8 Diskusion

In diesem Kapitel wird zunächst die in dieser Arbeit konstruierte Lösung mit bestehenden State-of-the-Art Applikationen verglichen. Anschließend werden auf Basis der gewonnenen Erkenntnisse die Funktionalität und Praktikabilität der konstruierten Lösung diskutiert.

Vergleich mit Fuzz4All

Es wird zunächst die entwickelte Lösung im Vergleich zu Fuzz4All betrachtet, das bereits in Kapitel 3.4 vorgestellt wurde. Zur Wiederholung, Fuzz4All umfasst die Kernfunktionalitäten Autoprompting zur Erstellung von Prompts aus bereitgestellten Dokumenten und eine Fuzzing Loop zur kontinuierlichen Generierung neuer Funktionen die verwendet werden, um das SUT zu testen. Daher werden zunächst die Prozesse zur Erstellung der Prompts betrachtet. Die in dieser Arbeit vorgestellte Lösung verwendet im Gegensatz zu Fuzz4All keine LLMs zur Erstellung von Prompts. Bei Fuzz4All ist es daher möglich, beliebige Arten von Dokumenten, die das SUT beschreiben, zu verwenden, während meine konstruierte Lösung auf die Verwendung der OAS beschränkt ist. Zum anderen ist der manuelle Aufwand bei meiner Lösung erheblich größer, da zunächst eine Prompt-Vorlage erstellt werden muss. Dieser Prozess erfordert eine experimentelle Vorgehensweise, bei der die Modellausgaben überprüft und ggf. durch entsprechende Anpassungen optimiert werden muss. Hat man allerdings eine geeignete Vorlage erst einmal gefunden, kann diese auch wiederverwendet werden. In [Xia+23] wird auch betont, dass ihre Lösung sich durch die Verwendung von LLMs mit dem SUT weiterentwickelt, indem lediglich die bereitgestellten Dokumentationen aktualisiert werden müssen. Ähnliches kann somit auch von der hier vorgestellten Lösung in dieser Arbeit behauptet werden. Auch wenn man auf die OAS eingeschränkt ist, handelt es sich hierbei um ein Dokument dass das SUT beschreibt und üblicherweise mit dem SUT aktualisiert wird.

Bei der Fuzzin Loop von Fuzz4All werden dann kontinuierlich neue Programme vom LLM erzeugt, mit denen das SUT anschließend getestet wird. Hierfür wird auch der Prompt, der dafür verwendet wird, kontinuierlich aktualisiert. Üblicherweise umfassen Testsequenzen eine Vielzahl von Testfällen, um eine umfassende Abdeckung zu gewährleisten und eine zuverlässige Ausage über das getestete System treffen zu können, ähnlich wie bei REST-QA. Die Erstellung all dieser Testfälle ausschließlich durch ein LLM wäre mit erheblichem Zeitaufwand und einer hohen Rechenlast verbunden. Daher stellt meine Lösung mit der LLM-Komponente sogenannte Generatoren bereit, die für eine Operation mehrere zufällige Werte generieren soll. Weiterhin ist eine automatische Dokumentation der gefundenen Fehler, wie sie bei Fuzz4All erfolgt, um später zu überprüfen,

ob es sich um Fehler des SUT handelt, in meiner entwickelten Lösung nicht implementiert. Dadurch ist eine systematische Überprüfung der Fehler nach ihrer Entdeckung nicht möglich.

Vergleich mit REST-QA

In diesem Abschnitt wird das von mir entwickelte Test-Tool mit dem in der Arbeit von [Kis21] entwickelten Test-Tool REST-QA verglichen. In derselben Arbeit hat man OpenAPI-based Stateful Testing (ObST) [Kis22] für das Erstellen von Testfällen vorgestellt. Nach der Arbeit von [Kis21] werden bei REST-QA die folgenden Phasen ausgeführt, bevor die eigentlichen Tests am SUT beginnen können: die Analyse der OAS und die darauf basierende Erstellung einer Testkonfigurationsdatei sowie die Überprüfung dieser Datei durch eine Person. Die Überprüfung der Testkonfigurationsdatei wird hierbei als notwendig bezeichnet, „[. . .] da nicht garantiert werden kann, dass die automatisierte Analyse der OAS zu einem korrekten und vollständigen Ergebnis führt“ [Kis21]. Vergleichsweise wäre bei der von mir konstruierten Lösung eine solche Überprüfung bei den Generatoren des Test-Tools notwendig, da diese mithilfe der LLM-Komponente erzeugt werden. Das Kapitel 7.1 zeigt schließlich, dass die LLMs häufig dazu neigten fehlerhaft Ergebnisse zu generieren.

REST-QA verwenden zudem konventionelle Methoden zur automatischen Wertegenerierung für die Properties. Dies ist jedoch nur möglich, wenn diese Properties „[...] in der OAS maschinenlesbar beschrieben werden oder die sich aus dem HTTP-Standard bzw. REST-Architekturstil ableiten lassen“ [Kis21]. Diese Methode ist dennoch grundsätzlich schneller und ressourcenschonender als wenn LLMs hierfür verwendet werden. Die in dieser Arbeit vorgestellte Softwarelösung zeigte bereits bei Properties, die auch durch konventionelle Verfahren hätten bearbeitet werden können, Schwierigkeiten. Es erweist sich zudem als unpraktisch, wenn die Gestaltungsmöglichkeiten der OAS weiter eingeschränkt werden müssen, da die verwendeten LLMs für bestimmte Schemata keine geeigneten Lösungen erzeugen konnten. Es erscheint daher sinnvoll, konventionelle Verfahren für bestimmte Properties weiterhin zu verwenden, während für Datentypen, bei denen diese Methoden nicht ausreichen, LLMs eingesetzt werden sollten.

In diesem Abschnitt werden die in Kapitel 5.3 beschriebenen Prozesse des Test-Tools mit den Prozessen der Test-Runner-Komponente von REST-QA verglichen, wie sie in der Arbeit [Kis21] vorgestellt werden. Der Fokus liegt dabei auf den Prozessen, die nach dem Empfangen der Antwort des SUTs ausgeführt werden, da diese den größten Unterschiede ausmachen. REST-QA besitzt eine sogenannte „Coverage-Tracker“-Komponente, die es ermöglich, „[. . .] die Abdeckung der Statuscodes pro Operation durch die Testfälle zu ermitteln“ [Kis21]. Wie in Ab-

bildung 10 dargestellt, sollte auch die in dieser Arbeit entwickelte Lösung über einen solchen Tracker verfügen. Diese Komponente ist jedoch in der praktischen Umsetzung nicht vorhanden. Aufgrund des Fehlens dieses Prozesses ist es auch nicht möglich, eine Untersuchung zur Abdeckung durchzuführen. Im nächsten Schritt aktualisiert REST-QA den aktuellen Zustand der Ressource und prüft, ob die Antwort des SUTs mit dem übereinstimmt, was als Property zu dieser Operation definiert wurde [Kis21]. Im Gegensatz dazu verfügt die von mir entwickelte Lösung über keine dieser genannten Prozesse. Daher ist auch eine kontinuierliche Überwachung des gesamten Testvorgangs durch eine Person erforderlich.

Weiterhin ist es möglich, REST-QA in einen typischen Softwareentwicklungsprozess mit Build-Automatisierung zu integrieren [Kis21]. Eine derartige Integration ist mit der von mir entworfenen Softwarelösung derzeit nicht möglich. Der Build-Automatisierungsprozess würde aktuell nur bis zur Erstellung des Test-Tools reichen. Danach wäre eine manuelle Überprüfung der Generatoren durch eine Person erforderlich. Darüber hinaus fehlen, wie bereits erwähnt, notwendige Funktionen zur automatischen Überprüfung der Ergebnisse.

Funktionalität & Praktikabilität

Abschließend zu diesem Kapitel soll in diesem Abschnitt noch die Funktionalität und Praktikabilität der konstruierten Lösung diskutiert werden. Wie aus Kapitel 7.2 hervorgeht, konnte das Test-Tool erfolgreich konstruiert und ausgeführt werden. Dabei wurden alle in der OAS definierten Operationen korrekt von der Web-API verarbeitet. Jedoch ist eine automatische Überprüfung und Dokumentation, wie bereits erwähnt, nicht möglich. Auch der in Kapitel 5.3 dargestellte Tracker ist in der praktischen Umsetzung nicht vorhanden. Solange diese Prozesse nicht implementiert sind, ist eine praktische Verwendung des Test-Tools zur Fehlerfindung in einem SUT mit unverhältnismäßigem Aufwand verbunden. Ein weiterer kritischer Aspekt ist die Erstellung der Generatoren mit der LLM-Komponente. Die Konstruktion einer geeigneten Prompt-Vorlage ist auch mit einem hohen Zeitaufwand verbunden. Zudem konnten während der Experimente keine durchgehend zuverlässigen Lösungen identifiziert werden, mit denen ein LLM stets zur OAS passende Generatoren erzeugt. Die Ursachen für fehlerhafte Modellausgaben konnten auch nicht immer eindeutig identifiziert werden. Es zeigte sich jedoch, dass LLMs für bestimmte Datentypen oder Schemata bessere Ergebnisse erzielen konnten als für andere. Es ist ratsam, Einschränkungen bei der Gestaltung der OAS zu vermeiden, um die Modellausgaben zu verbessern. Stattdessen sollte der Fokus auf der Modifikation der Prompt-Vorlage liegen. Letztlich sollte die OAS auch weiterhin so verwendet werden können, wie es auch vorgesehen ist. Allerdings ist das Ergebnis der Modellausgabe bei den konstruierten Prompt-Vorlagen noch sehr von der OAS und deren Inhalt abhängig. Um halbwegs geeig-

nete Generatoren zu erzeugen, war es auch entscheidend, dass die Properties in der OAS möglichst eindeutig definiert waren. Besonders Datentypen wie Integer mit klaren Grenzen durch die Schlüsselwörter `minimum` und `maximum` führten häufig zu Generatoren, die alle Kriterien erfüllten. Bei Datentypen wie Strings mit einem `date`-Format, wo Interpretationsspielraum bestand, war unklar, ob das aktuelle Datum oder ein zufälliges gewählt werden sollte. Aber auch Datentypen mit zu komplexen Patterns führten häufig zu fehlerhaften Generatoren.

9 Fazit

In dieser Arbeit wurde ein Konzept zur automatischen Erstellung von Testfällen für Web-APIs mithilfe vorhandener Large Language Models (LLMs) entwickelt. Auf dieser Grundlage wurde ein Prototyp auf einer geeigneten Plattform implementiert. Der Prototyp wurde anschließend hinsichtlich seiner Funktionalität und Praktikabilität evaluiert und mit bestehenden Lösungen verglichen. Dabei wurden einerseits Fuzz4All [Xia+23] untersucht, das LLMs zur Generierung von Testfällen einsetzt, und andererseits das in [Kis21] beschriebene Test-Tool REST-QA, das konventionelle Verfahren zur Erzeugung von Testfällen verwendet.

Es wird in dieser Arbeit vorgeschlagen, LLMs zur Generierung von Teilprogrammen für jede Operation in der OAS zu nutzen, die Eingabeparameter erfordert. Diese Teilprogramme, die auch als Generatoren bezeichnet werden, erzeugen dann zufällige Werte für die entsprechenden Operationen. Die Generatoren werden dann mit der Client-API kombiniert, die mithilfe von Swagger Codegen erstellt wurde, um ein Test-Tool zu erstellen. Die entwickelte Softwarelösung bietet zudem eine zentrale Konfigurationsmöglichkeit, die es ermöglicht, verschiedene Parameter eines Modells anzupassen und dadurch auch unterschiedliche Dekodierstrategien über eine Konfigurationsdatei auszuwählen und zu steuern. Darüber hinaus kann das Modell für die kausale Sprachverarbeitung frei ausgewählt werden, und es besteht die Möglichkeit, die Präzision zu wählen, mit der das Modell geladen werden soll. Zusätzlich kann auch eine Device-Map ausgewählt werden, um Modelle zu nutzen, die aufgrund ihrer Größe nicht vollständig von der Grafikkarte geladen werden können.

Die Konstruktion geeigneter Prompts, die es ermöglichen, über ein LLM adäquate Generatoren zu erzeugen, ist allerdings mit einem erheblichem Aufwand verbunden. Dies liegt teilweise daran, dass die Optimierung der Modellausgaben experimentell erfolgen muss, und es ungewiss ist, ob die vorgenommenen Anpassungen das Ergebnis letztendlich verbessern oder verschlechtern. Es wurde deswegen eine Struktur zur Gestaltung des Prompts entwickelt, um bei Anpassungen eine systematische Vorgehensweise zu gewährleisten. Dabei werden auch verschiedene Prompt-Engineering-Techniken berücksichtigt und angewendet.

In dieser Arbeit wurden Kriterien für die Optimierung der Modelle festgelegt, die sicherstellen sollen, dass die Funktionalität und Integrationsfähigkeit des Generators im Test-Tool gewährleistet sind. Die Testergebnisse haben gezeigt, dass drei der vier definierten Kriterien mit einer hoher Zuverlässigkeit erfüllt werden konnten. Häufig konnten dadurch Generatoren erzeugt werden, die im Test-Tool inte-

griert werden konnten, jedoch war es nicht immer möglich, Werte innerhalb des definierten Bereichs zu generieren. Auch Generatoren für Properties, die durch konventionelle Methoden hätten erstellt werden können, erwiesen sich häufig als fehlerhaft. Daher wird eine manuelle Überprüfung und gegebenenfalls Anpassung als erforderlich betrachtet. Eine klare und eindeutige Definition der Properties in der OAS war zudem ebenfalls entscheidend, um möglichst korrekte Generatoren zu erstellen. Die Ergebnisse verdeutlichen, dass konventionelle Verfahren, wie sie beispielsweise in der Arbeit von [Kis21] vorgestellt wurden, sich besser für einfache Properties eignen. Dennoch halte ich den Einsatz von LLMs für Properties, bei denen konventionelle Mittel nicht ausreichen, weiterhin für sinnvoll und als eine wertvolle Ergänzung. Jedoch ist es erforderlich, zunächst einen geeigneten Prompt zu finden, der ohne Einschränkungen an der OAS mit einem passenden Sprachmodell Generatoren erzeugen kann, die alle definierten Kriterien erfüllen. Diese Aufgabe konnte in dieser Arbeit nicht vollständig gelöst werden.

Aufgrund der fehlenden Implementierung wichtiger Prozesse konnte das konstruierte Test-Tool nicht vollständig evaluiert werden. Dennoch konnte gezeigt werden, dass semantisch korrekte Anfragen an das SUT gesendet und die entsprechenden Antworten empfangen werden konnten. Um den tatsächlichen Nutzen des vorgestellten Verfahrens zu bewerten, wäre es allerdings notwendig, die fehlenden Prozesse zu ergänzen und an verschiedenen Webservices zu testen. Obwohl die Lösung aufgrund der fehlenden Prozesse derzeit nicht für eine Build-Automatisierung geeignet ist, halte ich den Einsatz von LLMs in agilen Softwareentwicklungsprozessen grundsätzlich für machbar, abhängig von der Umsetzung und den verfügbaren Ressourcen. Im Gegensatz zur hier vorgestellten Softwarelösung verwendet beispielsweise Fuzz4All das LLM nahezu durchgängig, einschließlich der Generierung der einzelnen Testfälle. Eine solche Umsetzung wäre mit einem erheblichen zeitlichen Aufwand und einer höheren Rechenlast verbunden. Falls erforderlich, könnte das LLM auch über eine Schnittstelle von einem externen System bereitgestellt werden, das über die erforderlichen Ressourcen verfügt.

10 Ausblick

Das in dieser Arbeit entwickelte Test-Tool kann grundsätzlich um die Fähigkeit erweitert werden, die während einer Testsequenz durchgeführten Testfälle zu dokumentieren und diese mit den in der OAS definierten Properties abzugleichen. Dies soll eine automatische Erkennung von Verstößen gegen die OAS ermöglichen, wie es auch in der Arbeit von [Kis21] umgesetzt wurde. Auch die Ergänzung von Authentifizierungsverfahren würde sich als nützlich erweisen, insbesondere wenn die konstruierte Lösung an anderen Webservices wie der GitLab-API getestet werden soll.

Darüber hinaus muss auch noch ein geeigneter Prompt entwickelt werden, mit dem das LLM zuverlässig passende Generatoren erzeugen kann. In diesem Zusammenhang sollte auch das Verfahren zur Optimierung der Modellausgabe weiter ausgebaut werden. Da die manuelle Überprüfung sehr zeitaufwändig ist, sollte ein automatisiertes Verfahren, wie beispielsweise in der Arbeit von [Che+21] vorgestellt, implementiert werden. Es erscheint durchaus machbar, die in dieser Arbeit verwendeten Kriterien Ausführbarkeit (exe), korrekter Rückgabewert (crv) und korrekter Funktionsname (cfn) automatisiert zu überprüfen. Das Kriterium der Zufälligkeit (rnd) müsste jedoch mit vorangefertigten Unit-Tests überprüft werden. Die Ergebnisse müssten anschließend nur noch miteinander verglichen werden.

Es sollte auch in Betracht gezogen werden, dass LLMs beim Testen von Web-APIs nicht nur zur Erstellung von Werten für die Operationen verwendet werden können, sondern möglicherweise auch zur Analyse und Bewertung der Ergebnisse. Die Arbeit von [Ma+23] hat gezeigt, dass LLMs in der Lage sind, Bewertungen zu erstellen, die sogar besser sein können als die von menschlichen Experten. In diesem Zusammenhang könnte ein Verfahren entwickelt werden, bei dem die vom Test-Tool dokumentierten Testsequenzen, in denen ein Fehler entdeckt wurde, nachträglich mittels eines LLMs ausgewertet werden. Wie in Kapitel 2.3 ausgeführt, muss die Ursache des Fehlers derzeit noch von einer Person ermittelt werden. Es ist denkbar, dass dieser Prozess durch den Einsatz von LLMs beschleunigt werden könnte.

Anlage A: Quellen- und Literaturverzeichnis

[AGP19] Vaggelis Atlidakis, Patrice Godefroid und Marina Polishchuk. “RESTler: Stateful REST API Fuzzing”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Mai 2019. DOI: 10.1109/icse.2019.00083.

[AHS85] David H. Ackley, Geoffrey E. Hinton und Terrence J. Sejnowski. “A learning algorithm for boltzmann machines”. In: *Cognitive Science* 9.1 (1985), S. 147–169. ISSN: 0364-0213. DOI: https://doi.org/10.1016/S0364-0213(85)80012-4. URL: https://www.sciencedirect.com/science/article/pii/S0364021385800124.

[Aiy+23] Rachith Aiyappa u. a. “Can we trust the evaluation on ChatGPT?” In: *Proceedings of the 3rd Workshop on Trustworthy Natural Language Processing (TrustNLP 2023) (July 2023) 47-54* (März 2023). DOI: 10.18653/v1/2023.trustnlp-1.5. arXiv: 2303.12767 [cs.CL].

[All+23] Loubna Ben Allal u. a. “SantaCoder: don’t reach for the stars!” In: (Jan. 2023). DOI: 10.48550/ARXIV.2301.03988. arXiv: 2301.03988 [cs.SE].

[Aré22] José David Arévalo. *Creating and filling a Postgres DB with Docker Compose*. https://github.com/jdaarevalo/docker_postgres_with_data. 2022.

[Art+06] Thomas Arts u. a. “Testing telecoms software with quviq QuickCheck”. In: *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*. ICFP06. ACM, Sep. 2006. DOI: 10.1145/1159789.1159792.

[Bal+17] David Balduzzi u. a. “The Shattered Gradients Problem: If resnets are the answer, then what is the question?” In: *PMLR volume 70 (2017)* (Feb. 2017). DOI: 10.48550/ARXIV.1702.08591. arXiv: 1702.08591 [cs.NE].

[Ban+20] Kunal Banerjee u. a. “Exploring Alternatives to Softmax Function”. In: (Nov. 2020). DOI: 10.48550/ARXIV.2011.11538. arXiv: 2011.11538 [cs.LG].

[Ben+21] Emily M. Bender u. a. “On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?” In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. FAccT ’21. ACM, März 2021. DOI: 10.1145/3442188.3445922. URL: https://dl.acm.org/doi/pdf/10.1145/3442188.3445922.

[Bro+20] Tom Brown u. a. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems*. Hrsg. von H. Larochelle u. a. Bd. 33. Curran Associates, Inc., 2020, S. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[Bub+23] Sébastien Bubeck u. a. “Sparks of Artificial General Intelligence: Early experiments with GPT-4”. In: (März 2023). DOI: 10.48550/ARXIV.2303.12712. arXiv: 2303.12712 [cs.CL].

[CH00] Koen Claessen und John Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs”. In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. ICFP00. ACM, Sep. 2000. DOI: 10.1145/351240.351266.

[Che+21] Mark Chen u. a. “Evaluating Large Language Models Trained on Code”. In: (Juli 2021). DOI: 10.48550/ARXIV.2107.03374. arXiv: 2107.03374 [cs.LG].

[Che+23] Banghao Chen u. a. “Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review”. In: (Okt. 2023). DOI: 10.48550/ARXIV.2310.14735. arXiv: 2310.14735 [cs.CL].

[Cob+21] Karl Cobbe u. a. “Training Verifiers to Solve Math Word Problems”. In: (Okt. 2021). DOI: 10.48550/ARXIV.2110.14168. arXiv: 2110.14168 [cs.LG].

[Din+23] Jiayu Ding u. a. “LongNet: Scaling Transformers to 1,000,000,000 Tokens”. In: (Juli 2023). DOI: 10.48550/ARXIV.2307.02486. arXiv: 2307.02486 [cs.CL].

[DN84] Joe W. Duran und Simeon C. Ntafos. “An Evaluation of Random Testing”. In: *IEEE Transactions on Software Engineering* SE-10.4 (Juli 1984), S. 438–444. ISSN: 0098-5589. DOI: 10.1109/tse.1984.5010257.

[DS17] Rahul Dey und Fathi M. Salem. “Gate-variants of Gated Recurrent Unit (GRU) neural networks”. In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, Aug. 2017. DOI: 10.1109/mwscas.2017.8053243.

[FB97] George Fink und Matt Bishop. “Property-based testing: a new approach to testing for assurance”. In: *ACM SIGSOFT Software Engineering Notes* 22.4 (Juli 1997), S. 74–80. ISSN: 0163-5948. DOI: 10.1145/263244.263267.

[FG17] Jessica Ficler und Yoav Goldberg. “Controlling Linguistic Style Aspects in Neural Language Generation”. In: (Juli 2017). DOI: 10.48550/ARXIV.1707.02633. arXiv: 1707.02633 [cs.CL].

[FLD18] Angela Fan, Mike Lewis und Yann Dauphin. “Hierarchical Neural Story Generation”. In: (Mai 2018). DOI: 10.48550/ARXIV.1805.04833. arXiv: 1805.04833 [cs.CL].

[Geh+17] Jonas Gehring u. a. “Convolutional sequence to sequence learning”. In: *International conference on machine learning*. PMLR. 2017, S. 1243–1252.

[GLT21] Andrea Galassi, Marco Lippi und Paolo Torroni. “Attention in Natural Language Processing”. In: *IEEE Transactions on Neural Networks and Learning Systems* 32.10 (Okt. 2021), S. 4291–4308. ISSN: 2162-2388. DOI: 10.1109/tnnls.2020.3019893.

[Gol+24] Harrison Goldstein u. a. “Property-Based Testing in Practice”. In: (Apr. 2024). URL: https://harrisongoldste.in/papers/icse24-pbt-in-practice.pdf.

[Gor+14] Vladimír Gorej u. a. *Swagger Codegen*. https://github.com/swagger-api/swagger-codegen. 2014.

[Gun+23] Suriya Gunasekar u. a. "Textbooks Are All You Need". In: (Juni 2023). DOI: 10.48550/ARXIV.2306.11644. arXiv: 2306.11644 [cs.CL].

[Guo+24] Daya Guo u. a. "DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence". In: (Jan. 2024). DOI: 10.48550/ARXIV.2401.14196. arXiv: 2401.14196 [cs.SE].

[Ham04] Paul Hamill. *Unit test frameworks: tools for high-quality software development*. O'Reilly Media, Inc., 2004.

[Ham94] Richard Hamlet. "Random testing". In: *Encyclopedia of software Engineering* 2 (1994), S. 971–978. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=da250b154e113ce5a995be3d8c26c0a0c40abe6e.

[Hen+20] Dan Hendrycks u. a. "Measuring Massive Multitask Language Understanding". In: (Sep. 2020). DOI: 10.48550/ARXIV.2009.03300. arXiv: 2009.03300 [cs.CY].

[HHP20] Ahmad Hazimeh, Adrian Herrera und Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3 (Nov. 2020), S. 1–29. ISSN: 2476-1249. DOI: 10.1145/3428334.

[Hol+19] Ari Holtzman u. a. "The Curious Case of Neural Text Degeneration". In: (Apr. 2019). DOI: 10.48550/ARXIV.1904.09751. arXiv: 1904.09751 [cs.CL].

[HS97] Sepp Hochreiter und Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), S. 1735–1780. ISSN: 1530-888X. DOI: 10.1162/neco.1997.9.8.1735.

[HSS15] Kazuyuki Hara, Daisuke Saito und Hayaru Shouno. "Analysis of function of rectified linear unit used in deep learning". In: *2015 International Joint Conference on Neural Networks (IJCNN)*. IEEE, Juli 2015. DOI: 10.1109/ijcnn.2015.7280578.

[HT90] D. Hamlet und R. Taylor. "Partition testing does not inspire confidence (program testing)". In: *IEEE Transactions on Software Engineering* 16.12 (1990), S. 1402–1411. ISSN: 0098-5589. DOI: 10.1109/32.62448.

[htt] al pal (https://stats.stackexchange.com/users/276244/al-pal). *What is the role of feed forward layer in Transformer Neural Network architecture?* Cross Validated. (version: 2020-09-03). URL: https://stats.stackexchange.com/q/485910.

[Hu+15] Xiaoguang Hu u. a. "Improved beam search with constrained softmax for nmt". In: *Proceedings of Machine Translation Summit XV: Papers*. 2015. URL: https://aclanthology.org/2015.mtsummit-papers.23.pdf.

[Hug22a] Hugging Face. *The Hugging Face Course, 2022*. https://huggingface.co/course. [Online; accessed 17.06.2024]. 2022. URL: https://huggingface.co/course.

[Hug22b] Hugging Face. *The Hugging Face Docs, 2022*. https://huggingface.co/docs. [Online; accessed 17.06.2024]. 2022. URL: https://huggingface.co/docs.

[Kad+23] Jean Kaddour u. a. "Challenges and Applications of Large Language Models". In: (Juli 2023). DOI: 10.48550/ARXIV.2307.10169. arXiv: 2307.10169 [cs.CL].

[Kaz19] Amirhossein Kazemnejad. "Transformer Architecture: The Positional Encoding". In: *kazemnejad.com* (2019). URL: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/.

[Kis21] Benjamin Kissmann. "Automatisiertes Testen von RESTful Webservices zur Validierung von Claim-basierten Berechtigungskonzepten mittels der OpenAPI-Dokumentation". de. In: (2021). DOI: 10.25673/37346.

[Kis22] Benjamin Kissmann. *ObST - OpenAPI-based Stateful Testing*. https://github.com/HOME-programming-pub/ObST. 2022.

[Koc+22] Denis Kocetkov u. a. "The Stack: 3 TB of permissively licensed source code". In: (Nov. 2022). DOI: 10.48550/ARXIV.2211.15533. arXiv: 2211.15533 [cs.CL].

[Koj+22] Takeshi Kojima u. a. "Large Language Models are Zero-Shot Reasoners". In: (Mai 2022). DOI: 10.48550/ARXIV.2205.11916. arXiv: 2205.11916 [cs.CL].

[Kul+19] Sumith Kulal u. a. "Spoc: Search-based pseudocode to code". In: *Advances in Neural Information Processing Systems* 32 (2019).

[Lac+20] Marie-Anne Lachaux u. a. "Unsupervised Translation of Programming Languages". In: (Juni 2020). DOI: 10.48550/ARXIV.2006.03511. arXiv: 2006.03511 [cs.CL].

[Lee+23] Nayoung Lee u. a. "Teaching Arithmetic to Small Transformers". In: (Juli 2023). DOI: 10.48550/ARXIV.2307.03381. arXiv: 2307.03381 [cs.LG].

[Lew+22] Aitor Lewkowycz u. a. "Solving Quantitative Reasoning Problems with Language Models". In: *Advances in Neural Information Processing Systems*. Hrsg. von S. Koyejo u. a. Bd. 35. Curran Associates, Inc., 2022, S. 3843–3857. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/18abbeef8cfe9203fdf9053c9c4fe191-Paper-Conference.pdf.

[LHE21] Stephanie Lin, Jacob Hilton und Owain Evans. "TruthfulQA: Measuring How Models Mimic Human Falsehoods". In: (Sep. 2021). DOI: 10.48550/ARXIV.2109.07958. arXiv: 2109.07958 [cs.CL].

[Li+23] Raymond Li u. a. "StarCoder: may the source be with you!" In: (Mai 2023). DOI: 10.48550/ARXIV.2305.06161. arXiv: 2305.06161 [cs.CL].

[LKM22] Yaniv Leviathan, Matan Kalman und Yossi Matias. "Fast Inference from Transformers via Speculative Decoding". In: (Nov. 2022). arXiv: 2211.17192 [cs.LG].

[Log+21] Robert L. Logan u. a. “Cutting Down on Prompts and Parameters: Simple Few-Shot Learning with Language Models”. In: (Juni 2021). DOI: 10.48550/ARXIV.2106.13353. arXiv: 2106.13353 [cs.CL].

[Loz+24] Anton Lozhkov u. a. “StarCoder 2 and The Stack v2: The Next Generation”. In: (Feb. 2024). DOI: 10.48550/ARXIV.2402.19173. arXiv: 2402.19173 [cs.SE].

[Lu+21] Yao Lu u. a. “Fantastically Ordered Prompts and Where to Find Them: Overcoming Few-Shot Prompt Order Sensitivity”. In: (Apr. 2021). DOI: 10.48550/ARXIV.2104.08786. arXiv: 2104.08786 [cs.CL].

[Luo+19] Ling Luo u. a. “Unsupervised Neural Aspect Extraction with Sememes.” In: *IJCAI*. 2019, S. 5123–5129. URL: https://www.ijcai.org/proceedings/2019/0712.pdf.

[LY18] Yang Li und Tao Yang. “Word embedding for understanding natural language: a survey”. In: *Guide to big data applications* (2018), S. 83–104.

[Ma+23] Yecheng Jason Ma u. a. “Eureka: Human-Level Reward Design via Coding Large Language Models”. In: (Okt. 2023). DOI: 10.48550/ARXIV.2310.12931. arXiv: 2310.12931 [cs.RO].

[May+20] Joshua Maynez u. a. “On Faithfulness and Factuality in Abstractive Summarization”. In: (Mai 2020). DOI: 10.48550/ARXIV.2005.00661. arXiv: 2005.00661 [cs.CL].

[MHC19] David MacIver, Zac Hatfield-Dodds und Many Contributors. “Hypothesis: A new approach to property-based testing”. In: *Journal of Open Source Software* 4.43 (Nov. 2019), S. 1891. ISSN: 2475-9066. DOI: 10.21105/joss.01891. URL: https://joss.theoj.org/papers/10.21105/joss.01891.pdf.

[Mik+13] Tomas Mikolov u. a. “Efficient Estimation of Word Representations in Vector Space”. In: (Jan. 2013). DOI: 10.48550/ARXIV.1301.3781. arXiv: 1301.3781 [cs.CL].

[MJ+01] Larry R Medsker, Lakhmi Jain u. a. “Recurrent neural networks”. In: *Design and Applications* 5.64-67 (2001), S. 2.

[MOM23] Arseny Moskvichev, Victor Vikram Odouard und Melanie Mitchell. “The ConceptARC Benchmark: Evaluating Understanding and Generalization in the ARC Domain”. In: *Transactions on Machine Learning Research, 8/2023* (Mai 2023). DOI: 10.48550/ARXIV.2305.07141. arXiv: 2305.07141 [cs.LG].

[Pan99] Jiantao Pan. “Software testing”. In: *Dependable Embedded Systems* 5.2006 (1999), S. 1. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=28abbfdcd695f6ffc18c5041f8208dcfc8810aaf.

[Pap+19] Mike Papadakis u. a. “Mutation Testing Advances: An Analysis and Survey”. In: *Advances in Computers*. Elsevier, 2019, S. 275–378. DOI: 10.1016/bs.adcom.2018.03.015.

[Pat+23] Rajvardhan Patil u. a. “A Survey of Text Representation and Embedding Techniques in NLP”. In: *IEEE Access* 11 (2023), S. 36120–36146. ISSN: 2169-3536. DOI: 10.1109/access.2023.3266377.

[PI18] Goran Petrović und Marko Ivanković. “State of mutation testing at google”. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE ’18. ACM, Mai 2018. DOI: 10.1145/3183519.3183521.

[Qui+24] Federico Quin u. a. “A/B testing: A systematic literature review”. In: *Journal of Systems and Software* 211 (Mai 2024), S. 112011. ISSN: 0164-1212. DOI: 10.1016/j.jss.2024.112011.

[RAO92] Debra J. Richardson, Stephanie Leif Aha und T. Owen O’malley. “Specification-based test oracles for reactive systems”. In: *Proceedings of the 14th international conference on Software engineering*. 1992, S. 105–118.

[Ren+20] Shuo Ren u. a. “CodeBLEU: a Method for Automatic Evaluation of Code Synthesis”. In: (Sep. 2020). DOI: 10.48550/ARXIV.2009.10297. arXiv: 2009.10297 [cs.SE].

[RM21] Laria Reynolds und Kyle McDonell. “Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm”. In: (Feb. 2021). DOI: 10.48550/ARXIV.2102.07350. arXiv: 2102.07350 [cs.CL].

[Roz+23] Baptiste Rozière u. a. “Code Llama: Open Foundation Models for Code”. In: (Aug. 2023). DOI: 10.48550/ARXIV.2308.12950. arXiv: 2308.12950 [cs.CL].

[Sak+19] Keisuke Sakaguchi u. a. “WinoGrande: An Adversarial Winograd Schema Challenge at Scale”. In: (Juli 2019). DOI: 10.48550/ARXIV.1907.10641. arXiv: 1907.10641 [cs.CL].

[Sha+17] Louis Shao u. a. “Generating High-Quality and Informative Conversation Responses with Sequence-to-Sequence Models”. In: (Jan. 2017). DOI: 10.48550/ARXIV.1701.03185. arXiv: 1701.03185 [cs.CL].

[Shi+23] Jessica Shi u. a. “Etna: An Evaluation Platform for Property-Based Testing (Experience Report)”. In: *Proceedings of the ACM on Programming Languages* 7.ICFP (Aug. 2023), S. 878–894. ISSN: 2475-1421. DOI: 10.1145/3607860.

[SMK22] Ananya B. Sai, Akash Kumar Mohankumar und Mitesh M. Khapra. “A Survey of Evaluation Metrics Used for NLG Systems”. In: *ACM Computing Surveys* 55.2 (Jan. 2022), S. 1–39. ISSN: 1557-7341. DOI: 10.1145/3485766.

[Su+22] Yixuan Su u. a. “A Contrastive Framework for Neural Text Generation”. In: (Feb. 2022). DOI: 10.48550/ARXIV.2202.06417. arXiv: 2202.06417 [cs.CL].

[Swa24] Swagger. *OpenAPI Guide*. https://swagger.io/docs/specification/about/. [Online; accessed 17.06.2024]. 2024. URL: https://swagger.io/docs/specification/about/.

[Tay+21] Yi Tay u. a. “Charformer: Fast Character Transformers via Gradient-based Subword Tokenization”. In: (Juni 2021). DOI: 10.48550/ARXIV.2106.12672. arXiv: 2106.12672 [cs.CL].

[Van23] David Van Buren. “Guided scenarios with simulated expert personae: a remarkable strategy to perform cognitive work”. In: (Juni 2023). DOI: 10.48550/ARXIV.2306.03104. arXiv: 2306.03104 [cs.HC].

[Vas+17] Ashish Vaswani u. a. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Hrsg. von I. Guyon u. a. Bd. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[Vij+16] Ashwin K Vijayakumar u. a. “Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models”. In: (Okt. 2016). DOI: 10.48550/ARXIV.1610.02424. arXiv: 1610.02424 [cs.AI].

[WC20] Yu-An Wang und Yun-Nung Chen. “What Do Position Embeddings Learn? An Empirical Study of Pre-Trained Language Model Positional Encoding”. In: (Okt. 2020). DOI: 10.48550/ARXIV.2010.04903. arXiv: 2010.04903 [cs.CL].

[Wei23] Ariel Weinberger. *Pezzo*. https://github.com/pezzolabs/pezzo. 2023.

[WK92] Jonathan J Webster und Chunyu Kit. “Tokenization as the initial phase in NLP”. In: *COLING 1992 volume 4: The 14th international conference on computational linguistics*. 1992.

[WP21] Albert Webson und Ellie Pavlick. “Do Prompt-Based Models Really Understand the Meaning of their Prompts?” In: (Sep. 2021). DOI: 10.48550/ARXIV.2109.01247. arXiv: 2109.01247 [cs.CL].

[Wu+23] Skyler Wu u. a. “Analyzing Chain-of-Thought Prompting in Large Language Models via Gradient-based Feature Attributions”. In: (Juli 2023). DOI: 10.48550/ARXIV.2307.13339. arXiv: 2307.13339 [cs.CL].

[Xia+23] Chunqiu Steven Xia u. a. “Fuzz4All: Universal Fuzzing with Large Language Models”. In: (Aug. 2023). DOI: 10.48550/ARXIV.2308.04748. arXiv: 2308.04748 [cs.SE].

[Xu+24] Ruijie Xu u. a. “Benchmarking Benchmark Leakage in Large Language Models”. In: (Apr. 2024). DOI: 10.48550/ARXIV.2404.18824. arXiv: 2404.18824 [cs.CL].

[Yan+19] Min Yang u. a. “Exploring Human-Like Reading Strategy for Abstractive Text Summarization”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33.01 (Juli 2019), S. 7362–7369. ISSN: 2159-5399. DOI: 10.1609/aaai.v33i01.33017362.

[YJ18] ShumingShi YanSong und AI JingLi Tencent. “Joint learning embeddings for Chinese words and their components via ladder structured networks”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artifificial Intelligence (IJCAI-18)*. 2018, S. 4375–4381. URL: https://www.ijcai.org/Proceedings/2018/0608.pdf.

[Zel+19] Rowan Zellers u. a. “HellaSwag: Can a Machine Really Finish Your Sentence?” In: (Mai 2019). DOI: 10.48550/ARXIV.1905.07830. arXiv: 1905.07830 [cs.CL].

[ZH02] A. Zeller und R. Hildebrandt. “Simplifying and isolating failure-inducing input”. In: *IEEE Transactions on Software Engineering* 28.2 (2002), S. 183–200. ISSN: 0098-5589. DOI: 10.1109/32.988498. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=988498.

[Zha+22] Zhuosheng Zhang u. a. “Automatic Chain of Thought Prompting in Large Language Models”. In: (Okt. 2022). DOI: 10.48550/ARXIV.2210.03493. arXiv: 2210.03493 [cs.CL].

[Zha+23] Zheng Zhang u. a. “VISAR: A Human-AI Argumentative Writing Assistant with Visual Programming and Rapid Draft Prototyping”. In: UIST ’23 (16. Apr. 2023). DOI: 10.1145/3586183.3606800. arXiv: 2304.07810 [cs.HC].

[Zho+22] Hattie Zhou u. a. “Teaching Algorithmic Reasoning via In-context Learning”. In: (Nov. 2022). DOI: 10.48550/ARXIV.2211.09066. arXiv: 2211.09066 [cs.LG].

[Zho+23] Kun Zhou u. a. “Don’t Make Your LLM an Evaluation Benchmark Cheater”. In: (Nov. 2023). DOI: 10.48550/ARXIV.2311.01964. arXiv: 2311.01964 [cs.CL].

Anlage B: Abbildungsverzeichnis

Anlage C: Listing-Verzeichnis

Anlage D: Tabellenverzeichnis

Anlage E: Modellparameter

```yaml
# The batch size defines the number of samples that will be propagated.
# This value should always remain 1 and the "phase-ii -> loops" should be
# changed instead. This was implemented so that the execution time per sample
# can be recorded. A value greater than 1 would lead to longer generation
# times, as more samples are generated, but only the first sample generated
# is saved as a file.
batch-size: 1
# Activating sampling means randomly picking the next word. If set to False
# greedy decoding is used.
do-sample:
# Number of beams for beam search. Must be between 1 and infinity. 1 means no
# beam search.
num-beams:
# Number of groups to divide num_beams into in order to ensure diversity
# among different groups of beams that will be used by default in the
# generate method of the model. 1 means no group beam search.
num-beam-groups:
# Value to control contrastive search. 0 <= penalty_alpha <= 1
penalty-alpha:
# Value to control diversity for group beam search. that will be used by
# default in the generate method of the model. 0 means no diversity penalty.
# The higher the penalty, the more diverse are the outputs.
diversity-penalty:
# A higher temperature value leads to a more random output, while a lower
# value makes the output more deterministic.
temperature:
# top-k sampling limits the selection of the model to the k most probable
# tokens at each step of the output generation. Set 0 to deactivate.
top-k:
# Instead of sampling only from the most likely K words, in Top-p sampling
# chooses from the smallest possible set of words whose cumulative
# probability exceeds the probability p activate Top-p sampling by setting
# 0 < top_p < 1.
top-p:
# Maximum number of tokens that can be generated.
max-new-tokens: 1024
```

Listing 7: Ausschnitt zu den Modellparametern aus der Konfigurationsdatei

Anlage F: Programmablaufplan

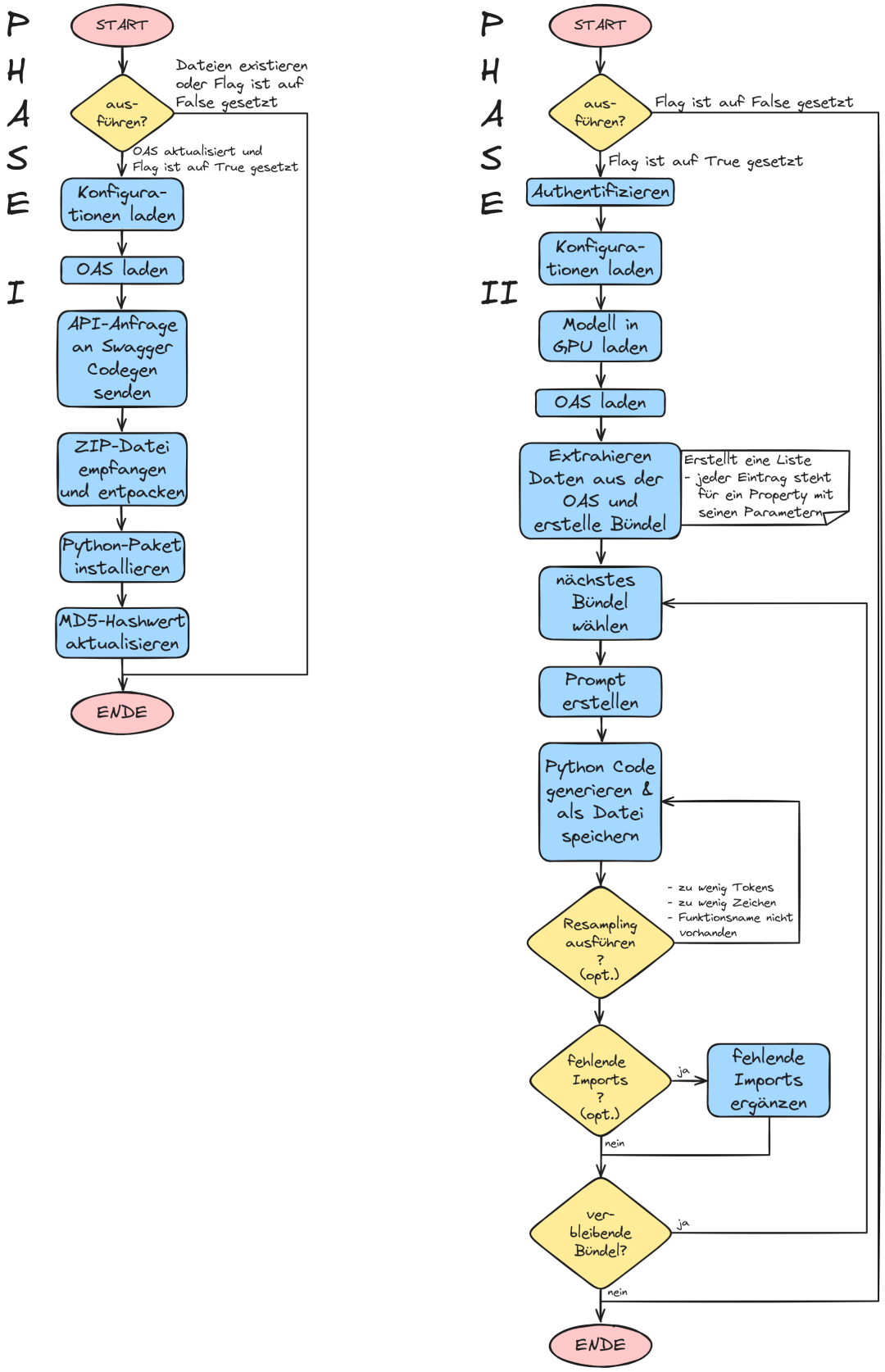


Abbildung 15: Flussdiagramme - Phase I und Phase II

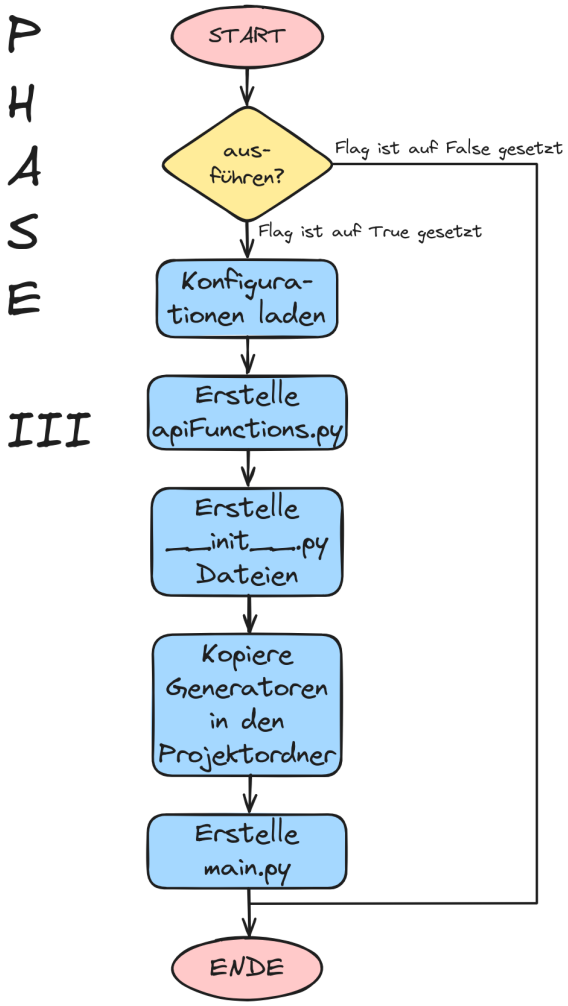


Abbildung 16: Flussdiagramme - Phase III

Anlage G: Diagramme

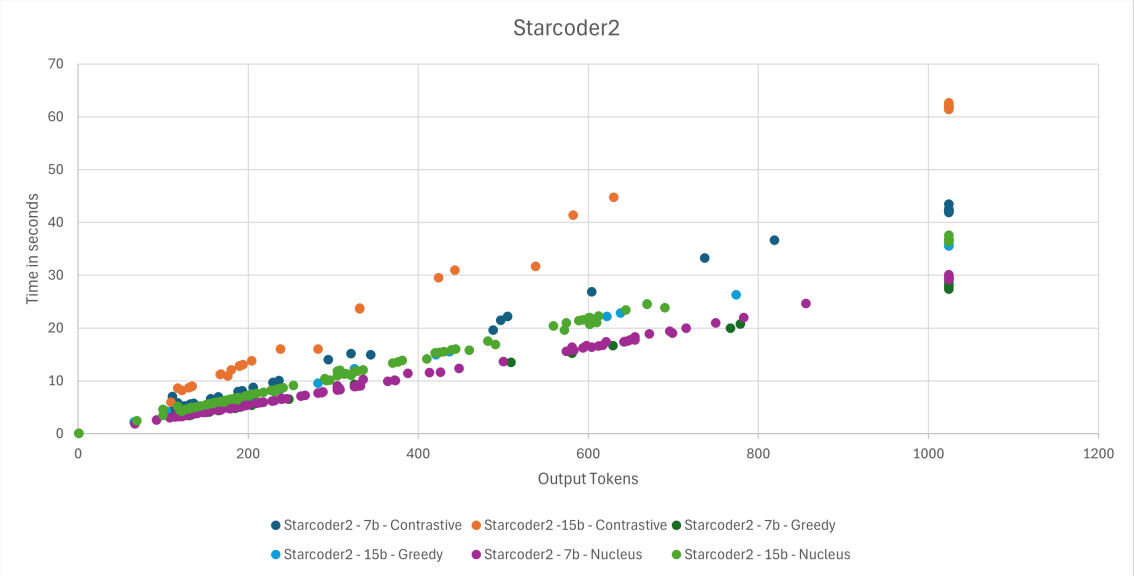


Abbildung 17: Starcoder2 - Generierungszeit im Verhältnis zur Anzahl der Ausgabe-Tokens

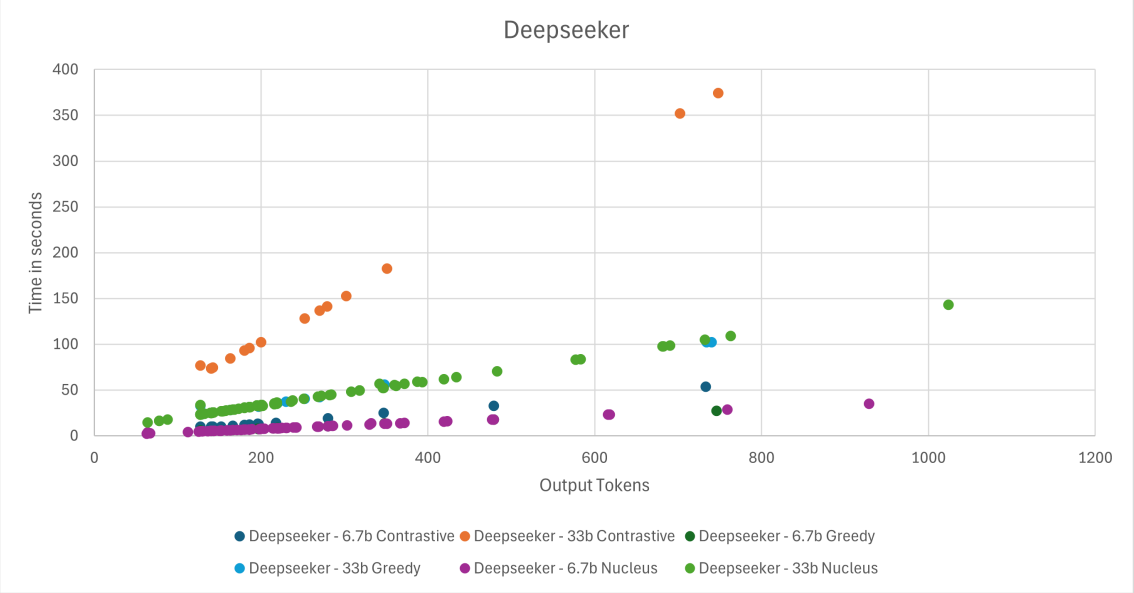


Abbildung 18: DeepSeeker - Generierungszeit im Verhältnis zur Anzahl der Ausgabe-Tokens

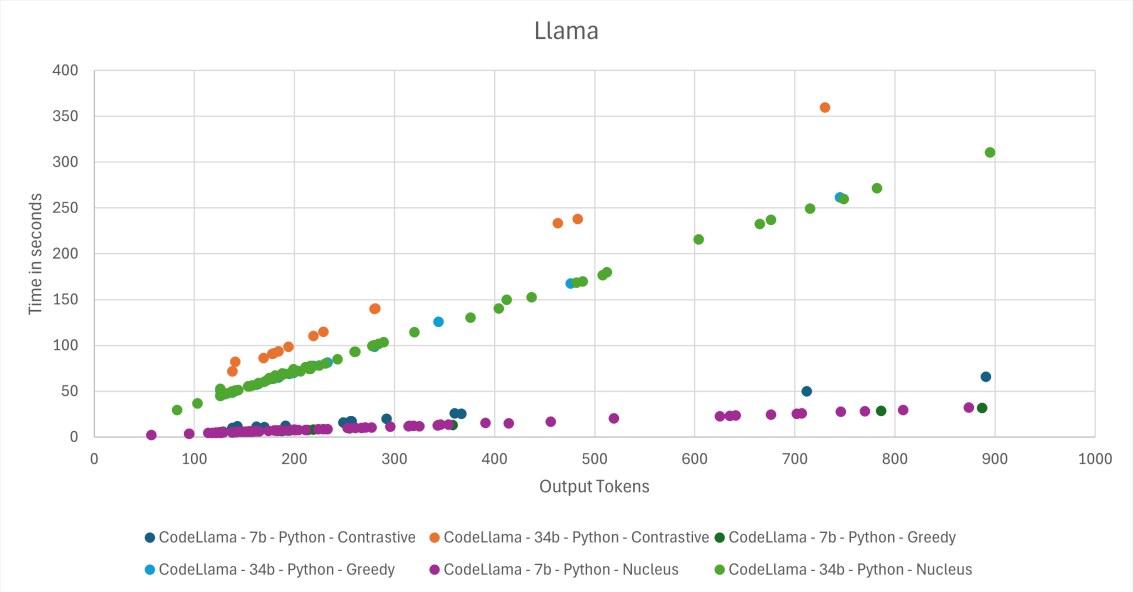


Abbildung 19: Code Llama für Python - Generierungszeit im Verhältnis zur Anzahl der Ausgabe-Tokens