Bachelorarbeit

des Studiengangs Informatik
der Dualen Hochschule Baden-Württemberg Mannheim

Thema

Vergleich von Methoden zur syntaktisch und semantisch korrekten Python-Codegenerierung durch Sprachmodelle

---

Jonas Putz

24. August 2026

Bearbeitungszeitraum:	02.06.26 - 25.08.26
Matrikelnummer, Kurs:	3369939, TINF23IKI1
Ausbildungsfirma:	Deutsches Zentrum für Luft- und Raumfahrt e.V.
Betrieblicher Betreuer:	Dominik Opitz
Gutachter der Dualen Hochschule:	Prof. Dr. Sudermann-Merx

Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem

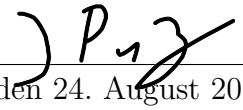
Thema

Vergleich von Methoden zur syntaktisch und semantisch korrekten Python-Codegenerierung durch Sprachmodelle

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.*

* falls beide Fassungen gefordert sind



Berlin, den 24. August 2026

Zusammenfassung

Die zunehmende Verbreitung generativer Sprachmodelle hat die automatische Codegenerierung in der Softwareentwicklung vorangetrieben. Trotz dieser Fortschritte weisen die erzeugten Programme häufig syntaktische und semantische Fehler auf, wie zum Beispiel bei der Nutzung externer Bibliotheken. Ziel dieser Bachelorarbeit ist es, die Wirksamkeit verschiedener Code-Generierungsmethoden zur Reduktion solcher Fehler zu evaluieren. Im Fokus stehen dabei die Methoden Temperature-Sampling, Sampling & Filtering, Post-Revising, AdapT, RoCode sowie ein neu entwickelter Context-Adaptive Sampler, der syntaktische Korrektheit durch statische Analyse und gezielte Token-Filterung garantiert.

Die Arbeit gliedert sich zunächst in theoretische Grundlagen zu Sprachmodellen und formalen Sprachen, anschließend in die Beschreibung der jeweiligen Generierungsmethoden, Implementierungsdetails und schließlich den experimentellen Aufbau auf dem HumanEval-Benchmark. Für vier Qwen-Modelle (500 Millionen bis 7 Milliarden Parameter) werden PassRate, Token-Verbrauch und Generierungszeit verglichen. RoCode erreicht mit $95,4\%$ PassRate die höchste Performance, während der Kontext adaptive Ansatz keine syntaktischen Fehler erzeugt und gleichzeitig die schnellste Generierungszeit erzielt. AdapT verbessert die PassRate im Durchschnitt um etwa $3,8\%$ gegenüber reinem Temperature-Sampling, während Sampling & Filtering und Post-Revising den größten Token-Verbrauch aufweisen.

Die Ergebnisse zeigen, dass eine Kombination aus statischer Analyse und Ausführungstests (RoCode) die beste Fehlerreduktion liefert, während der Context-Adaptive Sampler besonders effizient, modular und gezielt bestimmte Fehlerquellen eliminiert. Die Arbeit schließt mit einer Antwort auf die Forschungsfrage und Vorschläge für zukünftige Verbesserungen.

Abstract

The spread of large generative language models has accelerated automated code generation in software engineering. Nevertheless, generated programs frequently contain syntactic and semantic defects, for example when invoking external libraries. This bachelor thesis evaluates the effectiveness of serveral code-generation strategies in mitigating such errors. The study focuses on Temperature-Sampling, Sampling & Filtering, Post-Revising, AdapT, RoCode and a newly devised Context-Adaptive Sampler, that guarantees syntactic correctness through static analysis and targeted token filtering.

The work is structured in three parts. First, it revies the theoretical foundations of language models and formal programming languages. Second, it details each generation technique, including implementation specifics. Third, it describes the experimental setup on the HumanEval benchmark. Four Qwen models (500 million to 7 billion parameters) are compared across three matrics: pass-rate, token consumption and generation time. RoCode attains the highest performance with a 95.4% pass-rate, while the Context-Adaptive approach yields no syntactic errors and the fastest generation time. AdapT raises the average pass-rate by roughly $3,8\%$ relative to plain temperature sampling. Sampling & Filtering and Post-Revising exhibit the greatest token consumption.

The results demonstrate that combining static analysis with execution testing (RoCode) delivers the most substantial error reduction, whereas the Context-Adaptive Sampler offers superior efficiency, modularity and targeted elimination of specific error sources. The thesis concludes by addressing the research question and proposing directions for future enhancements.

Inhaltsverzeichnis

Abkürzungsverzeichnis

KI künstliche Intelligenz

DLR Deutschen Zentrum für Luft- und Raumfahrt e.V.

LM Sprachmodell (engl. Language Model)

BPE Byte-Pair Encoding

AST abstrakter Syntaxbaum

BFCL Berkeley Function Calling Leaderboard

Tabellenverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

1 Einleitung

1.1 Motivation

Immer mehr Programme werden schon heute zu großen Teilen durch generative künstliche Intelligenz (KI) erstellt [1]. Hierbei wird häufig die Technik des Vibe-Coding verwendet, bei der ein Benutzer die Funktionen eines Programms in textueller Form beschreibt und ein Sprachmodell (engl. Language Model, LM) auf Grundlage dieser Eingabe den benötigten Programmcode autonom verfasst. Allerdings produzieren vor allem kleinere Sprachmodelle teilweise fehlerhaften Code, der nicht ausführbar ist oder logische Probleme beinhaltet [2, 3]. Im Institut für Softwaretechnologie am Deutschen Zentrum für Luft- und Raumfahrt e.V. (DLR) wird dieses Verhalten genauer untersucht und Methoden getestet, um derartige Fehler zu reduzieren.

Während logische Fehler meist auf bestimmte Codeabschnitte zugeschnittene Tests benötigen, um zuverlässig entdeckt zu werden, ist für anderen Fehlerarten eine statische Codeanalyse ausreichend. So können zum Beispiel syntaktische Fehler, die fehlerhafte Verwendung von Methodenschnittstellen und falsche Einbindungen externer Bibliotheken bereits vor der Ausführung des generierten Programmcodes markiert werden. Auf Grundlage einer solchen statischen Analyse können demnach bestimmte Fehlerquellen frühzeitig erkannt und behoben bzw. bereits im Vorhinein unterbunden werden.

Die Gruppe Intelligente Softwaresysteme am DLR beschäftigt sich mit dem Einsatz von KI in verschiedenen Bereichen. Im Zuge dessen sollen die Fähigkeiten verschiedener Ansätze zur Codegenerierung getestet, evaluiert und bewertet werden. Hierbei ist eine Software in der Programmiersprache Python entstanden, die mehrere State-of-the-Art-Methoden implementiert und deren Leistung im Bereich der Fehlervermeidung bewertet. Zugleich wird ein prototypischer Algorithmus vorgestellt, der eine hohe Anpassungsfähigkeit an eine Vielzahl möglicher Problemquellen demonstriert.

Ziel dieser Bachelorarbeit ist es, Methoden zur syntaktisch und semantisch korrekten Python-Codegenerierung durch Sprachmodelle zu vergleichen. Hierbei wird ein besonderer Fokus auf semantische Fehler gelegt, welche durch eine inkorrekte Nutzung von Schnittstellen entstehen. Insgesamt sollen Ansätze aufgezeigt werden, die den praktische Nutzen verschiedener LMs durch eine reduzierte Fehlerrate verbessern.

1.2 Problemstellung

Ein zentrales Problem für den weitreichenden Einsatz von LMs bei der Codegenerierung stellen die von diesen erzeugten syntaktischen und semantischen Fehler dar [2]. Insbesondere kommt es wiederholt vor, dass der Umgang mit externen Bibliotheken zu ungültigem Programmcode führt. So werden beispielsweise Methoden mit falschen Parametern aufgerufen oder nicht vorhandene Klassen aus Modulen importiert. Damit wird ersichtlich, dass selbst syntaktisch korrekter Code Probleme aufweist, die eine Ausführung bereits im Vorfeld verhindern.

Hieraus folgt, dass von LMs generierter Code häufig fehlerbelastet ist und manuelle Korrekturen erforderlich sind, was den durch den Einsatz von generativer KI möglichen Effizienzgewinn in der Softwareentwicklung einschränkt.

Da LMs keine interne Validierung für erzeugten Programmcode besitzen, ist es notwendig, erweiterte Ansätze zu entwickeln, die syntaktische und semantische Überprüfungen übernehmen. Aus diesem Hintergrund ergibt sich die zentrale Forschungsfrage der vorliegenden Arbeit:

> Können Methoden zur Syntax- und API-gesteuerten Beschränkung bei der Token-Generierung die Fehlerrate von erzeugtem Python-Code signifikant reduzieren?

1.3 Vorgehensweise

Die vorliegende Arbeit ist in sechs Phasen gegliedert, die den Kapiteln entsprechen. Das Ziel besteht nicht darin, ein vollwertiges Produkt zu entwickeln, sondern die Funktionsweise und Effektivität unterschiedlicher Ansätze zur Reduktion von Fehlern bei der Codegenerierung zu vergleichen.

Hintergrund Hier werden die Basisinformationen dargelegt, die für ein umfassendes technisches Verständnis der in dieser Arbeit aufgezeigten Methoden erforderlich sind.

Methoden der Codegenerierung Im zweiten Schritt werden verschiedene Ansätze, darunter State-of-the-Art-Techniken sowie ein neues prototypisches Verfahren, vorgestellt und deren Funktionsweise erläutert. Dabei wird auch die Eignung der Ansätze für die feh-

lerfreie Codegenerierung mit Fokus auf Syntax und der Verwendung externer Programm-Bibliotheken analysiert.

Implementierungshinweise Im Anschluss erfolgt die praktische Umsetzung der zuvor etablierten Verfahren. In diesem Kapitel werden besondere Schwierigkeiten und Möglichkeiten bei der Implementierung beleuchtet, um Abweichungen zwischen den theoretisch konzipierten und den praktisch realisierten Funktionsweisen heraus zu arbeiten.

Evaluierung Um die theoretischen Überlegungen zu verifizieren wird im Kapitel Evaluierung ein Testaufbau skizziert. Anhand etablierter Testdaten werden die vorgestellten Methoden gegenübergestellt. Dabei werden etablierte Metriken auf die gewonnenen Daten angewandt.

Diskussion Schließlich werden die zuvor gesammelten Ergebnisse kritisch diskutiert. Hierbei werden unter anderem Unterschiede anhand der verwendeten LMs sowie Trade-Offs zwischen Geschwindigkeit, Qualität und Abhängigkeiten der Methoden betrachtet. Im Fazit wird die Forschungsfrage beantwortet und eine Zusammenfassung der gewonnenen Erkenntnisse geliefert.

Ausblick Zum Schluss wird ein Ausblick gegeben, wie die gewonnenen Erkenntnisse für weitere Forschung in diesem Bereich genutzt werden können.

2 Grundlagen

Dieses Kapitel beschreibt die theoretischen Grundlagen, die für das Verständnis der in dieser Arbeit beschriebenen Verfahren notwendig sind. Zunächst werden verwandte Publikationen betrachtet, die ähnliche Themen bearbeiten. Darauf werden die wesentlichen Verwendungszwecke und Komponenten von Sprachmodellen erläutert, beginnend mit der Tokenisierung, der Architektur von LMs und schließlich verschiedenen Generationstechniken. Anschließend wird das Konzept formaler Sprachen behandelt, wobei ein besonderer Fokus auf Python als formale Sprache gelegt wird. Hierzu zählen die Rollen von Pasern und Syntaxbäumen sowie die Unterscheidung zwischen Syntax und Semantik, die beide für die Erkennung und Prävention von Fehlern in generiertem Code verwendet werden können.

2.1 Verwandte Arbeiten

In einem Paper aus dem Jahr 2021 evaluieren Mitarbeiter von OpenAI ihr neues Codex-Modell anhand eines manuell dafür erstellten Test-Datensatzes (vgl. [4]). Dieser dient seitdem der Bewertung von Sprachmodellen bei der Generierung von Python-Code in verschiedenen Arbeiten (vgl. [5, 6]). Das Codex-Modell selbst wurde explizit auf Python-Code trainiert[4] und stellt somit einen frühen Ansatz des Code Fine-Tunings für die autonome Programmierung dar. Ein ähnlicher Fine-Tuning Ansatz wurde 2023 für das Gorilla-Modell verwendet [7]. Forschende konnten dabei demonstrieren, dass die Leistung bei dem Formulieren von API-Calls durch gezieltes Fine-Tuning sowie Document-Retrieving signifikant verbessert werden kann [7]. Diese Erkenntnisse implizieren, dass Fehler bei der Verwendung externer Bibliotheken durch spezielles Training und die Bereitstellung relevanter Schnittstellen-Informationen vermieden werden können. Somit liefert diese Arbeit eine potenzielle Lösung für die zuvor formulierte Problemstellung in Bezug auf die Fehlervermeidung bei externen Bibliotheken, wobei jedoch keine direkte Beschränkung der Modell-Ausgabe erfolgt, sondern eine besondere Informationsumgebung und ein spezielles Modelltraining verwendet werden. Die allgemeine Vermeidung von Python-Fehlern bei der Codegenerierung wurde unter anderem 2025 durch den RoCode-Ansatz untersucht [5]. Diese Methode nutzt verschiedene Analysetools zur autoregressiven Entwicklung von Programmen. Werden Fehler erkannt, wird ein Rollback ausgelöst und der betroffene Code

neu generiert. Dieser Ansatz verringert die Fehlerquote, indem nach Durchführung des Rollbacks die Wahrscheinlichkeit der Tokens, die zu dem Fehler geführt haben, reduziert wird.

2.2 Sprachmodelle

Sprachmodelle (engl. Language Models, LMs) haben sich in den letzten Jahren als zentraler Baustein für die natürliche Sprachverarbeitung etabliert [8]. Ihr wesentlicher Nutzen ergibt sich aus der Fähigkeit, mit hoher Wahrscheinlichkeit die nächste Sequenz von Tokens vorherzusagen, wodurch sie für eine Vielzahl von Aufgaben eingesetzt werden können. So erstrecken sich die Anwendungen von LMs von der Text- und Codegenerierung über die maschinelle Übersetzung und automatisierten Frage-Antwort-Systemen bis hin zur Klassifikation. Während bei der Text-Generierung Sprachmodelle meist direkt ohne weitere Zusatzanwendungen verwendet werden, zeigen maschinelle Übersetzungen, Frage-Antwort-Systeme und Klassifikationsaufgaben die Flexibilität dieser Modelle.

Ein Übersetzungssystem verwendet häufig eine abgewandelte Architektur, die aus zwei Bereichen besteht: Der Erste verarbeitet den Eingabetext und unterstützt den Zweiten bei der Erstellung des Übersetzungstextes mit inhaltlichen Informationen [9]. Frage-Antwort-Systeme, die auf eine breit aufgestellte Wissensbasis firmeninterner Dokumente zurückgreifen, nutzen hingegen häufig externe Anwendungen in einer sogenannten RAG-Pipeline für die Wissensabfrage [10]. Klassifikationsaufgaben wiederum verwenden die rohen Daten der internen Netzwerke des LM, um beispielsweise Themen zu erkennen oder E-Mails als Spam zu markieren. Die Flexibilität, bereits vortrainierte Modelle mit wenigen Änderungen für unterschiedliche Aufgaben einzusetzen, macht LMs besonders attraktiv für Forschung und Praxis.

In allen diesen Anwendungsbereichen teilen LMs die grundlegende Annahme, dass die Sprache auf statistischen Mustern basiert, die durch KI erfasst werden können. In den folgenden Kapiteln werden die technischen Grundlagen, die diese Fähigkeiten ermöglichen, erläutert.

2.2.1 Tokenisierung

Tokenisierung bildet den ersten Schritt und dient der Übersetzung von rohen Text- bzw. Code-Sequenzen in eine maschinenlesbare Form. Im Kontext von Sprachmodellen werden hierbei Strategien vor allem durch die Granularität der Unterteilung der Eingabe und wie die Tokens anschließend mit numerischen Identifikatoren verknüpft werden, entschieden.

Eine Wort-Level-Tokenisierung betrachtet jedes Wort als eigenständigen Token. Dieses intuitive Vorgehen stößt jedoch auf zwei wesentliche Einschränkungen: Erstens ist das resultierende Vokabular zwangsläufig sehr groß, weil sämtliche Wörter abgedeckt werden müssen. Zweitens können neu gebildete Worte, etwa Variablennamen in der Programmierung, nicht verarbeitet werden, was zu Problemen führen kann [11].

Byte-Pair Encoding (BPE) beginnt mit einem Basislexikon, das sämtliche Einzelzeichen enthält. Durch wiederholtes Ersetzen der am häufigsten vorkommenden Zeichenpaare entstehen allmählich längere Token, bis die gewünschte Lexikongröße erreicht ist [11]. Das BPE-Konzept wird durch SentencePiece von der Zeichenebene auf die Byte-Ebene erweitert, sodass auch beliebige Unicode-Texte verarbeitet werden können [12].

Nachdem ein Vokabular vollständig mit Tokens gefüllt ist, wird jedem Token eine eindeutige Identifikator-ID zugewiesen. Diese numerischen Darstellungen werden anschließend durch eine Embedding-Matrix in hochdimensionale Vektoren, die Embeddings, umgewandelt. Diese dienen nicht nur der Identifikation der Tokens, sondern vor allem der Darstellung der dadurch ausgedrückten semantischen Information, beispielsweise der Bedeutung eines Wortes [13].

2.2.2 Modellarchitektur

Die Architekturen moderner LMs basieren überwiegend auf der Transformer-Architektur [14]. Ursprünglich bestehen Transformer-Modelle aus einer Reihe identischer Blöcke, die jeweils ein Self-Attention-Modul und ein positionsbasiertes Feed-Forward-Netzwerk enthalten. Diese erlauben es, Abhängigkeiten zwischen Tokens unabhängig von ihrer Entfernung zu erfassen und gleichzeitig die Reihenfolge der Eingabesequenz zu berücksichtigen [15].

Weiter lassen sich Transformer-Modelle in Decoder-Only- und Encoder-Decoder-Architekturen unterteilen. Bei erstem werden die für jeden Token zugänglichen Informationen ausschließlich auf die vorhergehenden Tokens beschränkt. Dadurch entsteht ein kausaler Kontext, der für generative Aufgaben, also dem Vorhersagen der nächsten Tokens, erforderlich ist. Damit eignen sich Decoder-Only-Architekturen besonders für Aufgaben wie Text- bzw. Codegenerierung, da sie die Sprache sequentiell modellieren [16].

Im Gegensatz dazu bestehen Encoder-Decoder-Modelle aus zwei getrennten Netzwerken. Der Encoder verarbeitet die gesamte Eingabesequenz und identifiziert, ähnliche wie in Decoder-Only-Architekturen, wichtige Abhängigkeiten auf Grundlage von Position und Inhalt einzelner Tokens. Da die Eingabesequenz bereits vollständig bekannt ist, kann der Encoder auf Informationen zukünftiger Tokens zugreifen, was ihn von Decoder-Only-Modellen unterscheidet. Der Decoder übernimmt schließlich die Aufgabe der Ausgabe-Generierung, wobei er die vom Encoder gelieferten Kontext-Vektoren nutzt. Diese Architektur hat sich insbesondere in Aufgaben wie der maschinellen Übersetzung bewährt, da sie ein Verständnis der Eingabe und bisherigen Ausgabe erfordert [9].

2.2.3 Generierungsmethoden

In der Textgenerierung durch Sprachmodelle lassen sich drei Haupttypen von Decodierungsstrategien unterscheiden: deterministische Verfahren, probabilistische Sampling-Strategien und Constraint-basierte Decodierung. Diese Ansätze unterscheiden sich nicht nur in ihrer algorithmischen Umsetzung, sondern auch hinsichtlich ihrer Auswirkungen auf die Qualität, Vielfalt und Fehlertoleranz des generierten Codes.

Deterministische Verfahren Deterministische Decodierungsalgorithmen wählen bei jedem Schritt entsprechend einer nicht stochastischen Methode aus. `Greedy decoding` ist hierbei die einfachste Variante: Der Token mit der höchsten Wahrscheinlichkeit wird ausgewählt und das Modell mit diesem fortgesetzt. Dieses Auswahlverfahren wird in Gleichung 1 dargestellt, wobei t_{next} der vorhergesagte Token, V das Vokabular und p_t die durch das Sprachmodell berechnete Wahrscheinlichkeit des Tokens t ist. Dieser Ansatz ist extrem schnell, erfordert keinen zusätzlichen Speicheraufwand und lieft immer dasselbe Ergebnis, kann jedoch leicht zu suboptimalen Lösungen führen, wenn ein lokales Maximum die wahre globale Lösung unterdrückt.

$$t_{\text{next}} = \arg\max_{t \in V} p_t \tag{1}$$

Ein Schritt weiter geht `Beam-search`, wobei für jeden Zeitschritt die k Sequenzen mit höchster Wahrscheinlichkeit gespeichert werden. Am Ende des Generierungsprozesses wird die Sequenz mit der höchsten Gesamtwahrscheinlichkeit ausgewählt. `Beam-search` verbessert die Qualität im Vergleich zu `Greedy decoding`, indem es eine begrenzte Menge von Pfaden simultan verfolgt, erhöht jedoch dadurch den Rechenaufwand. Insbesondere für lange Sequenzen muss daher eine Balance zwischen Qualität und Effizienz gefunden werden [17].

Probabilistische Sampling-Strategien Probabilistische Decodierungsmethoden führen durch gezielte Stochastik zu mehr Vielfalt in der erzeugten Sequenz. Sie reduzieren die Gefahr, dass das Modell in einem lokalen Maximum stecken bleibt und generieren teils kreativer wirkende Lösungen.

- **Temperature-Sampling**: Hier wird ein Temperatur-Parameter T verwendet, der durch eine Skalierung die Wahrscheinlichkeitsverteilung glättet oder verbreitert. Eine hohe Temperatur ($T > 1$) erzeugt ein flacheres Spektrum, was zu mehr Zufälligkeit führt; bei niedrigen Temperaturen ($T < 1$) wird die Verteilung schärfer und das Modell tendiert eher zu deterministischen Ergebnissen [18]. Hierbei wird die Wahrscheinlichkeit der Auswahl von Tokens, wie in Gleichung 2 gezeigt, modifiziert.

$$\begin{aligned} q_t &= \frac{\exp(p_t/T)}{\sum_{v \in V} \exp(p_v/T)} \\ t_{\text{next}} &\sim q_t,\ t \in V \end{aligned} \tag{2}$$

- **Top-k-Sampling**: Hier werden die Ausgaben auf die k wahrscheinlichsten Tokens beschränkt, bevor deren Wahrscheinlichkeiten normalisiert werden. `Top-k` verhindert somit, dass extrem seltene Tokens ausgewählt werden [19, 20].

- **Top-p-Sampling**: Statt einer festen Tokenanzahl wird hier anhand der kumulativen Wahrscheinlichkeiten eingeschränkt. So wird die Menge der Tokens auf die minimale Auswahl begrenzt, die zusammen eine Wahrscheinlichkeit von p (z.B. 90%)

abdeckt. Dadurch wird das Sampling dynamisch an die Verteilung angepasst und verhindert, dass seltene, aber sehr unwahrscheinliche Tokens ausgewählt werden, während gleichzeitig die Modellflexibilität erhalten bleibt [21, 20].

Constraint-basierte Decodierung Um die Wahrscheinlichkeit von syntaktisch oder semantisch fehlerhaften Ausgaben zu verringern, werden Decodierungsprozesse mit zusätzlichen Beschränkungen versehen. Diese können zum Beispiel auf einer expliziten Grammatik oder auf einem Datenstruktur-Schema beruhen. Der Decoder limitiert hierbei für jedes Ausgabetoken die möglichen Kandidaten auf diejenigen, die zuvor definierten Regeln entsprechen [22]. Diese Technik ist besonders geeignet, wenn syntaktische Korrektheit der Rückgabe höchste Priorität hat. Nachteile sind die Notwendigkeit, für jeden Anwendungsfall eine Grammatik zu definieren und eine durch die Validierung sinkende Performance [22, 23]. Alternativ wird häufig auch eine Beschränkung auf eine definierte Datenstruktur verwendet. So erlauben beispielsweise LlamaCPP und Ollama, zwei weit verbreitete Anwendungen für die Ausführung von Sprachmodellen, die Angabe eines JSON-Schemas, welches bei der Generierung erfüllt wird [24, 25].

Kombination und praktische Aspekte In vielen realen Anwendungen werden die oben beschriebenen Ansätze kombiniert, um ein Gleichgewicht zwischen Qualität, Vielfalt und Korrektheit zu erreichen. So wendet LlamaCPP standardmäßig `Top-k`, `Top-p`, `Temperature-Sampling` und vier weitere Strategien nacheinander an, um einen nächsten Token auszuwählen [26]. Diese Kombination zeigt, dass die richtige Abstimmung verschiedener Ansätze erforderlich ist, um ein gewünschtes Ergebnis zu erzielen.

2.3 Formale Sprachen

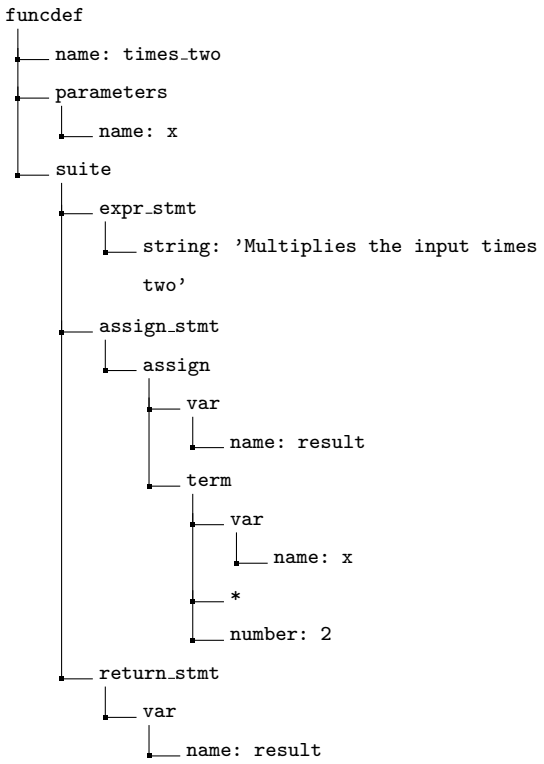
Eine formale Sprache ist ein abstraktes Konzept, das die Grundlage für die formale Beschreibung von Programmiersprachen, Grammatikregeln und Automatentheorie bildet. Sie wird mathematisch als Menge von Zeichenketten *strings* definiert, die durch ein endliches Alphabet Σ erzeugt werden können. Σ ist hierbei eine endliche Menge von Symbolen und ein *string* eine endliche Folge von Symbolen aus Σ oder die leere Zeichenkette ϵ [27]. Formale Sprachen werden in der Regel durch grammatische Regeln erzeugt, die erlaubte Ableitungen von Nichtterminalsymbolen zu Terminalen beschreiben.

Der klassische Ansatz zur Kategorisierung formaler Sprachen ist die Chomsky-Hierarchie, die von Noam Chomsky in den 1950er Jahren entwickelt wurde. Sie gliedert Sprachen in vier Typen, die sich in ihrer Ausdruckskraft und ihren Berechenbarkeits-Eigenschaften unterscheiden [28, 29]:

- **Reguläre Sprachen** (Typ 3): Diese Sprachen lasse sich durch endliche Automaten, reguläre Ausdrücke oder rechtslineare grammatische Regeln beschreiben. Sie besitzen die geringste Ausdruckskraft, sind jedoch vollständig berechenbar und können in linearer Zeit verarbeitet werden. Typische Beispiele sind Token-Muster in Lexern oder einfache Dateipfade.
- **Kontextfreie Sprachen** (Typ 2): Diese Sprachen werden durch kontextfreie Grammatiken beschrieben, bei denen jede Regel die Form $A \rightarrow \alpha$ hat, wobei A ein Nichtterminal und α eine beliebige Folge von Terminalen und Nichtterminalen ist. Kontextfreie Sprachen sind mächtiger als reguläre Sprachen, können jedoch nicht immer alle grammatischen Konstruke einer Programmiersprache modellieren.
- **Kontextsensitive Sprachen** (Typ 1): Diese Sprachen erfordern kontextsensitive Grammatiken, bei denen die Regeln die Form $\alpha A \beta \rightarrow \alpha \gamma \beta$ haben, wobei A ein Nichtterminal und α, β, γ beliebige Zeichenketten sind. Kontextsensitive Sprachen sind in ihrer Ausdruckskraft stärker als kontextfreie Sprachen, können jedoch nicht von deterministischen endlichen Automaten erkannt werden. Sie sind in der Praxis selten für die Syntaxanalyse von Programmiersprachen anzutreffen, werden jedoch in Spezialfällen wie der Analyse von Sprachen mit Speicherbeschränkungen verwendet.
- **Rekursiv aufzählbare Sprachen** (Typ 0): Der allgemeinste Typ in der Hierarchie, der den von Turing-Maschinen erkennbaren Sprachen entspricht. Jede Sprache, die von einer Turing-Maschine akzeptiert wird, gehört zu dieser Klasse, und umgekehrt. Rekursiv aufzählbare Sprachen besitzen die größte Ausdruckskraft, sind jedoch im Allgemeinen nicht entscheidbar. Es gibt demnach keine allgemeingültige Möglichkeit, zu entscheiden, ob ein gegebener *string* zur Sprache gehört.

Jede höhere Ebene der Hierarchie umfasst sämtliche Sprachen der darunterliegenden Ebenen, sodass reguläre Sprachen ein Spezialfall von kontextfreien und diese wiederum ein Spezialfall der kontextsensitiven und rekursiv aufzählbaren Sprachen sind. Diese Struktur ermöglicht es, die Komplexität und Berechenbarkeit einer formalen Sprache, wie zum

```python
def times_two(x):
    """
    Multiplies the input
       times two.
    """
    result = x * 2
    return result
```



Quellcode 1: Vergleich von Python-Code und Python-Syntax-Baum

Beispiel einer Programmiersprachen, systematisch zu bewerten.

2.3.1 Python als Formale Sprache

Python wird seit Einführung als dynamische, interpretierte Sprache mit einer klar definierten Grammatik beschrieben. Im Quellcode des CPython-Interpreters liegt hierfür eine Datei mit dem Namen `python.gram`[1], die eine nahezu vollständige, kontextfreie Beschreibung der Syntax liefert. Diese Grammatik wird durch die Parser-Engine des CPython-Interpreters genutzt, um aus dem Programmtext einen abstrakten Syntaxbaum (AST) zu generieren, wie es in Quellcode 1 dargestellt ist. Ergänzend dazu werden in PEP-Dokumenten[2] verschiedene Stilrichtlinien und Konventionen festgehalten, die zwar keine Grammatikregeln formulieren, aber einen geordneten Umgang mit der Sprache ermöglichen.

Die Grammatik von Python lässt sich als kontextfreie Grammatik interpretieren. Trotz der kontextsensitiven Behandlung bestimmter Sprachelemente, etwa der Whitespace-basierten Blockabgrenzung, lässt sich diese Beschränkung durch eine angepasste Lexer-Logik

[1] `https://github.com/python/cpython/blob/main/Grammar/python.gram`
[2] `https://peps.python.org/`

überwinden. So sind in der Python-Grammatik die Tokens `INDENT` und `DEDENT` genannt, die den Beginn, bzw. das Ende eines Blocks markieren [30]. Folglich ist ein Python-Parser verpflichtet, zusätzliche Constraints zu berücksichtigen, um entsprechend des aktuellen Einrückungslevels bei Bedarf `INDENT` oder `DEDENT` Tokens einzupflegen.

2.3.2 Syntax und Semantik

In der Analyse von Programmen bildet die Unterscheidung zwischen Syntax und Semantik ein grundlegendes Prinzip, das sich nicht nur auf die formale Beschreibung von Programmiersprachen, sondern auch auf die praktische Validierung von vom Sprachmodell generierten Code auswirkt. Der folgende Abschnitt erläutert zunächst die theoretische Basis dieser beiden Konzepte und zeigt anschließend, welche Werkzeuge eingesetzt werden können, um syntaktische und semantische Fehler zu erkennen und zu verhindern.

Syntax Die Syntax einer Programmiersprache beschreibt die formalen Regeln, die bestimmen, ob eine Zeichenkette zu einem gültigen Ausdruck, einer Anweisung oder einem gesamten Programm gehört. In der formalen Grammatik wird die Syntax üblicherweise als kontextfreie Grammatik formuliert, wobei Nichtterminalsymbole aufeinander folgen und sich in konkrete Token auflösen. Für Python ist die Syntax in der Datei `python.gram`[3] festgehalten und von CPython selbst als kontextfreie Grammatik interpretiert. Syntaxregeln legen ausschließlich die strukturelle Ordnung der Tokens fest und geben dabei keine Bedeutung für die ausgeführten Operationen vor. So wird zum Beispiel durch die Syntax die Verwendung eines Variablennamens bei Rechenoperationen erlaubt, allerdings keine Aussage darüber getroffen, ob dieser bereits im Vorfeld definiert wurde.

Zur automatisierten Syntaxprüfung können verschiedene Bibliotheken zur Umwandlung von Code in einen AST verwendet werden. In dieser Arbeit werden hierfür die Python-Standardbibliothek `ast`[4] und die Programm-Bibliothek `lark`[5] eingesetzt. Beide Tools implementieren die Python-Grammatik und können somit eine syntaktische Überprüfung durchführen.

[3] `https://github.com/python/cpython/blob/main/Grammar/python.gram`
[4] `https://docs.python.org/3/library/ast.html`
[5] `https://github.com/lark-parser/lark`

Semantik Die Semantik beschreibt, was ein syntaktisch korrektes Programm tatsächlich tut. Sie umfasst die Bedeutung von Ausdrücken, die Ausführung von Anweisungen, den Umgang mit Speicher, Typen und Nebenwirkungen. Somit sind sowohl logische Fehler, die sich in inkorrekten Ausgaben von Methoden ausdrücken, als auch viele Fehler, die während der Ausführung ausgelöst werden, wie zum Beispiel `ZeroDivisionError` oder `IndexError`, der Semantik zuzuordnen.

Zur semantischen Validierung werden meist gezielte Unit-Tests eingesetzt, da diese in der Lage sind, logische Fehler aufzudecken. Wird durch deren Ausführung der gesamte Code aufgerufen, können zudem ein Großteil der nicht durch Logikfehler entstandenen Probleme entdeckt werden.

3 Methoden der Codegenerierung

In diesem Kapitel werden sechs unterschiedliche Ansätze zur generativen Erstellung von Python-Code vorgestellt, die im Rahmen dieser Arbeit untersucht werden. Ein Basisansatz, der ausschließlich auf Temperature-Sampling basiert, sowie zwei einfache Variationen, die auf diesem Grundprinzip aufbauen. Der aktuelle Stand der Technik wird durch zwei Verfahren repräsentiert, die fortgeschrittene Mechanismen einsetzen, um die Fehleranfälligkeit zu reduzieren. Abschließend wird ein eigener prototypischer Ansatz vorgestellt, der speziell auf die Vermeidung syntaktischer Fehler und spezieller semantischer Probleme bei der Nutzung externer Bibliotheken ausgelegt ist. Für jeden Ansatz werden die Vorgehensweise der Codegenerierung und die theoretische Eignung für die Erzeugung fehlerfreier Programme erläutert.

Alle Methoden erhalten als Eingabe die Signatur einer zu implementierenden Methode sowie eine funktionsbeschreibende Dokumentation, die Beispiele enthält. Ziel ist es, Quellcode zu erzeugen, sodass die Funktion für die vorgesehene Testeingaben das korrekte Verhalten und die erwartete Ausgabe liefert.

3.1 Temperature-Sampling

Temperature-Sampling stellt ein simples Decodierungsverfahren dar, bei dem die Wahrscheinlichkeitsverteilung des Sprachmodells durch einen Temperatur-Parameter T moduliert wird. Durch Erhöhung der Temperatur wird die Verteilung flacher, wodurch die Wahrscheinlichkeit für seltenere Tokens steigt, während bei einer niedrigeren Temperatur die Verteilung schärfer wird und das Modell tendenziell deterministische Entscheidungen trifft. In der Praxis bedeutet dies, dass ein einzelner Prompt an das Modell übergeben wird und die Ausgabe Token-weise generiert wird, wobei zu jedem Schritt ein Sample aus der Temperatur-modifizierten Wahrscheinlichkeitsverteilung gezogen wird.

Dieser Ansatz ist bewusst naiv und dient als Baseline: er erfordert keinerlei zusätzliche Logik, Prüfungen oder Constraints, sondern verlässt sich ausschließlich auf die Fähigkeit des Modells, aus dem gegebenen Prompt einen plausiblen Code-Abschnitt zu erzeugen. In dieser Form liefert Temperature-Sampling keine Garantie für die Richtigkeit des resultierenden Codes. Der Ansatz dient daher primär als Referenzpunkt, gegen den komplexere, fehlervermeidende Verfahren evaluiert werden können. Temperature-Sampling wird in Ab-

bildung 1 schematisch dargestellt.

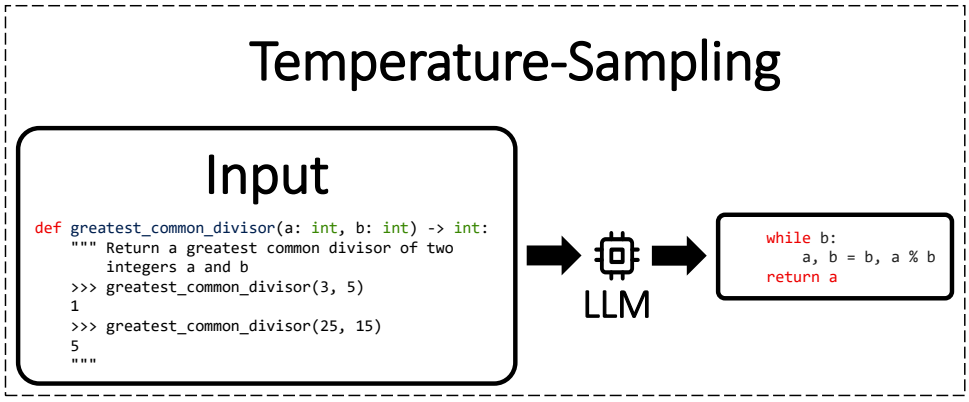


Abbildung 1: Temperature-Sampling Algorithmus

3.2 Sampling & Filtering

Auch bei diesem Ansatz wird der Code ausschließlich durch wiederholtes Temperature-Sampling erzeugt. Für jede vollständige Methode werden anhand von Testdaten mehrere Generationen ausprobiert. Erreicht die aktuelle Ausgabe kein positives Testergebnis, erhält das Sprachmodell einen erneuten Versuch zur Generierung einer korrekten Ausgabe. Dieser Prozess wird solange fortgeführt, bis die Methode frei von Fehlern ist. Zusammenfassend lässt sich die Vorgehensweise als wiederholtes Temperature-Sampling beschreiben, das bis zur Erzeugung fehlerfreien Codes anhält.

Durch das anschließende Filtering kann das Verfahren eine vollständige Fehlerfreiheit garantieren: bei jedem fehlgeschlagenen Test wird die betreffende Generation verworfen und ein neuer Sampling-Durchlauf gestartet. In der Praxis ist jedoch zu berücksichtigen, dass das wiederholte Sampling zu einem signifikanten Anstieg der Gesamt-Ausführungszeit führt und ein Limit an maximalen Versuchen die Garantie für Fehlerfreiheit verringert. Damit ist diese Methode oftmals ineffizient und für komplexere Programme nur schwer skalierbar. Sampling & Filtering wird in Abbildung 2 schematisch dargestellt.

3.3 Post-Revising

Post-Revising beginnt mit einem klassischen Temperature-Sampling-Ansatz zur Codegenerierung. Sobald die Zielmethode fertiggestellt ist, wird sie anhand vordefinierter Testfälle ausgeführt und etwaige Fehler protokolliert. Liegt ein Fehler oder ein inkorrektes Ergebnis vor, wird eine iterative Korrektur eingeleitet, bei der das Sprachmodell erneut den ur-

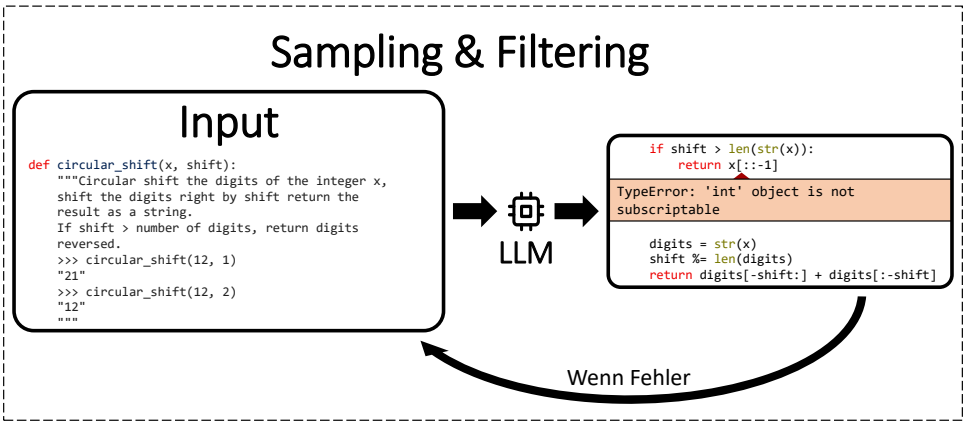


Abbildung 2: Sampling & Filtering Algorithmus

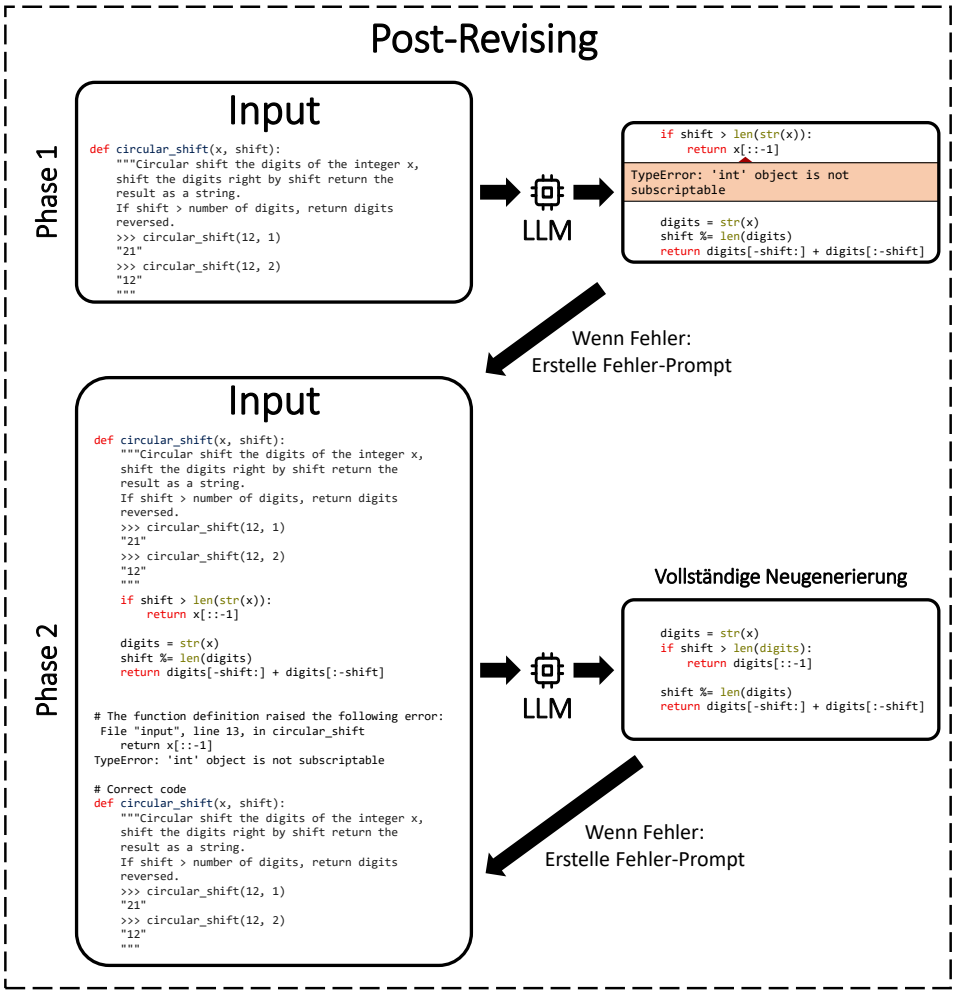


Abbildung 3: Post-Revising Algorithmus

sprünglichen Prompt, den bereits generierten Code und eine Fehlermeldung erhält. Durch diese Rückführung wird dem Modell gezieltes Feedback geliefert, das die Wahrscheinlichkeit für die Auswahl korrekter Tokens erhöhen soll (siehe Abbildung 3).

Diese Vorgehensweise garantiert die Fehlerfreiheit des Endcodes, da dieser systematisch durch Testfälle validiert und bei auftretenden Fehlern neu generiert wird. Allerdings kann die Notwendigkeit, bei jeder Fehlermeldung ein neues Sampling durchzuführen, erneut zu einer erheblichen Steigerung der Gesamt-Ausführungszeit führen, wodurch die Effizienz des Ansatzes beeinträchtigt wird.

Trotz dieser Ineffizienz kann Post-Revising gegenüber dem Sampling & Filtering Ansatz Vorteile bieten, da die Korrekturmechanik explizit auf Wissen über den konkreten Fehler zurückgreifen kann.

3.4 AdapT

Adapt[6] nutzt die Idee, den Temperatur-Parameter T bei der Token-Generierung dynamisch an die Wahrscheinlichkeit des wahrscheinlichsten Tokens anzupassen. Im Kern wird die Unsicherheit des Modells als Messgröße für kritische Eintscheidungsstellen im Code interpretiert. Hohe Unsicherheit bedeutet, dass das Sprachmodell mehrere plausible Tokens in Betracht zieht und damit „unsicher“ ist. In solchen Situationen wird die Temperatur T erhöht, um die Verteilung zu verbreitern und damit die Kreativität der Ausgabe zu fördern. Umgekehrt wird bei einer hohen Generationssicherheit im Modell, meist weil die Struktur des Codes eine klaren Pfad vorgibt, die Temperatur verringert, um Abweichungen von syntaktisch erwarteten Regeln zu minimieren [6].

Die Autoren des Papers stellen fest, dass hohe Unsicherheit in der Regel mit kritischen Entscheidungen des Modells zusammenhängt. Durch eine Analyse der generierten Codes aus einer umfangreichen Testreihe konnten sie zudem feststellen, dass diese kritischen punkte häufig an den Anfangsstellen von Codeblöcken liegen, etwa bei der Definition von Funktionskörpern oder Schleifen. Auf Basis dieser Erkenntnisse erhöht AdapT die Temperatur explizit für einen Token, sobald ein Codeblock beginnt, während ansonsten die Standard-Temperatur beibehalten wird [6] (vgl. Abbildung 4).

Obwohl AdapT die syntaktische Kohärenz des generierten Codes verbessern kann [6], bietet es keine Garantie für absolute Fehlerfreiheit. Das generierte Programm wird nicht aus-

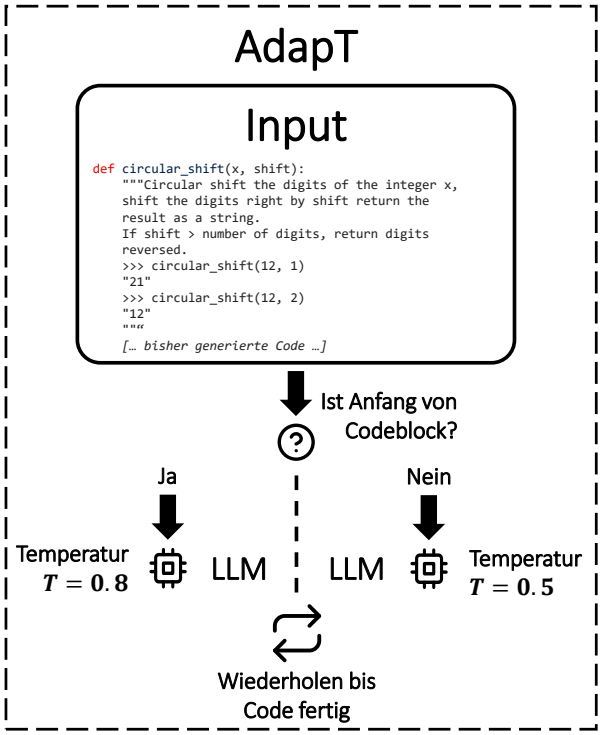


Abbildung 4: AdapT Algorithmus

geführt, sondern lediglich anhand der Temperatur-abhängigen Token-Verteilung erzeugt. Die Methode ist zudem besonders schnell, da sie als 1-Shot-Ansatz nicht auf wiederholte Rückfragen oder Iterationen des Sprachmodells angewiesen ist. In diesem Sinne stellt AdapT einen effizienten, aber begrenzten Ansatz dar, der primär auf der statistischen Unsicherheit des Modells basiert und nicht auf expliziten syntaktischen oder semantischen Prüfungen.

3.5 RoCode

RoCode[5] stellt einen progressiven Ansatz zur generativen Erstellung von Programmcode dar. Das Verfahren kombiniert ein klassisches Temperature-Sampling mit einer statischen Analyse, um syntaktische und semantische Fehler frühzeitig zu erkennen, und nutzt einen gezielten Rollback-Mechanismus, um fehlerhafte Token zu korrigieren. Dabei wird keine vollständige Neugenerierung des Codes ausgelöst. Stattdessen erfolgt lediglich ein Rückschritt bis zum Token mit der höchsten Entropie und die Wahrscheinlichkeiten für die zuvor fehlerhaften Tokens werden reduziert; Quasi ein Soft-Constraint-Decoding, das die Modellprobabilitäten lokal anpasst, ohne einzelne Tokens explizit direkt zu verbieten [5].

Im Detail wird der Code zunächst durch den Temperature-Sampling-Mechanismus generiert. Zu definierten Zeitpunkten wird der bislang erzeugte Code einem statischen Analysewerkzeug übergeben. Im Originalpapier wird hier ein Compiler als solches Tool vorgeschlagen. Das Werkzeug liefert anschließend Feedback zu bestimmten syntaktischen oder semantischen Fehlern. Beim Auftreten eines Fehlers wird der Code bis zum Token mit der höchsten Entropie zurückgesetzt. Danach wird die Wahrscheinlichkeit für die zuvor fehlerhaften Tokens in der Token-Verteilung reduziert, wobei insbesondere die des letzten, fehlerauslösenden Tokens am stärksten vermindert wird. In der darauf folgenden Sampling-Phase sinkt somit die Wahrscheinlichkeit für eine erneute Fehlerrate. Dieser Prozess wird solange wiederholt, bis die statische Analyse keine Fehler mehr meldet [5] (vgl. Abbildung 5).

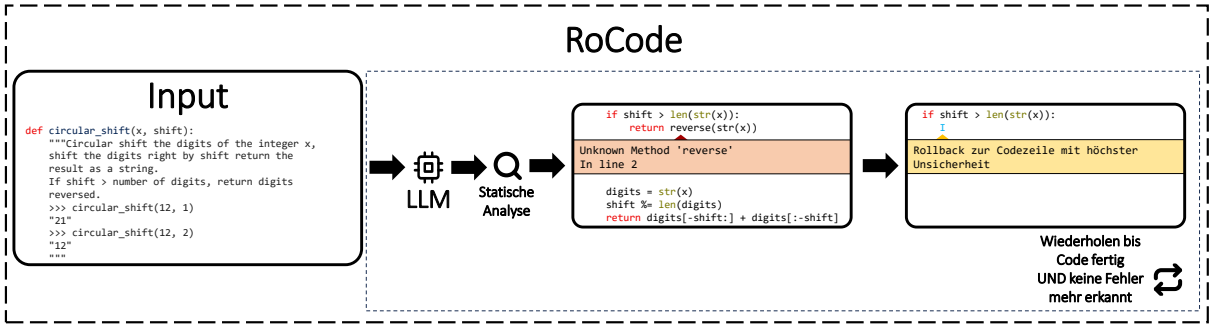


Abbildung 5: Verfeinfachte Darstellung des RoCode Algorithmus

RoCode garantiert die Fehlerfreiheit im Rahmen der durchgeführten Tests. Solange die eingesetzten statischen Analysetools keine Fehler melden, gilt der Code als gültig. Die Kombination aus Temperature-Sampling, statischer Analyse und gezieltem Rollback macht RoCode zu einem effizienten Mechanismus, der die Fehlerrate von generiertem Python-Code signifikant senkt, ohne die generative Flexibilität des Modells stark einzuschränken.

3.6 Context-Adaptive Sampler

Der Context-Adaptive Sampler ist ein eigenständiger Prototyp, der sich explizit auf die Vermeidung von Syntax- und Bibliotheksfehlern bei der Token-Generierung konzentriert. Im Gegensatz zu vergleichbaren Ansätzen verzichtet er bewusst auf die Ausführung des generierten Codes, da diese Vorgehensweise Nebenwirkungen mit sich bringen und die Effizienz stark beeinträchtigen würde. Stattdessen wird die Fehlerfreiheit in einem definierten Kontext garantiert, indem unmittelbar nach der Token-Generierung eine Reihe

von statischen Prüfungen ausgeführt wird.

Die Methode verfolgt einen modularen Aufbau, der die Erweiterung des Testkontexts erleichtert. Neue Prüfkriterien lassen sich einfach als weitere Module ergänzen, sodass die Lösung skalierbar bleibt. Gleichzeitig werden die Testmethoden nur bei Bedarf aktiviert, wodurch die Gesamteffizienz des Samplingprozesses nicht unnötig verlangsamt wird. Derzeit liegt der Fokus auf der syntaktischen Korrektheit sowie dem korrekten Einsatz externer Bibliotheken, wobei spätere Iterationen weitere semantische Prüfungen integrieren können.

3.6.1 Funktionsweise

Die Funktionsweise des Ansatzes gliedert sich in fünf zusammenhängende Phasen, die jeweils aufeinander aufbauen:

Zunächste erfolgt die Token-Generierung im Rahmen eines Constaint-Decodings. Hierbei werden Tokens, die bereits als Fehlerquellen identifiziert wurden, strikt verboten, während die übrigen Tokens über das gewohnte Temperature-Sampling erzeugt werden. Diese Kombination nutzt die Präzision von Constraints, um syntaktisch und semantisch problematische Tokens auszuschließen, ohne dabei die Vielfalt der generierten Ausgabe zu stark einzuschränken.

Anschließend wird immer ein Syntax-Check durchgeführt. Dieser prüft, ob die aktuell erzeugte Sequenz dem grammatischen Aufbau der Programmiersprache entspricht. Gleichzeitig wird dabei ein AST aufgebaut, der für die Identifikation weiterer Tests verwendet wird. Der Syntax-Check ist ein zentrales Element, das frühzeitig syntaktische Ungereimtheiten erkennt, bevor sie in weiterführende Prüfungen gelangen.

Aus dem erzeugten Syntaxbaum werden anschließend zusätzliche Checks abgeleitet, wobei hier jeder nur aktiviert wird, wenn für ihn relevante, noch nicht überprüfte, Code-Abschnitte entstanden sind. Zu diesen gehören im aktuellen Prototyp:

- **Import-Check**: Prüfung, ob importierte Module tatsächlich zugänglich sind und die gewünschten Klassen existieren.
- **Method Call-Check**: Validierung, ob bei aufgerufenen Methoden die übergebenen benannten Argumente tatsächlich angenommen werden.

Diese strukturierten Prüfungen ermöglichen eine gezielte Fehlerbehandlung, indem sie auf den semantischen Ebenen des Codes nach Inkonsistenzen suchen.

Sobald der Code ohne Fehler fertiggestellt und das Ende der Generation erreicht ist, wird final geprüft, ob die Syntax vollständig und nicht leer ist.

Treten bei diesen Prüfungen Fehler auf, wird der betroffene, sowie wenige diesem vorausgehende Tokes aus dem bislang generierten Code entfernt. Weiter wird die zum Fehler führende Sequenz abgespeichert, sodass bei einer erneuten Generation dieser der Fehlertoken aus dem Sampling entfernt werden kann.

Durch die Kombination von Constraint-Decoding, Syntax-Check, gezielten semantischen Prüfungen und dem abschließenden Vollständigkeitcheck entsteht ein robustes Verfahren, das syntaktische und besondere semantische Fehler vollständig ausschließen kann. Ein vereinfachter schematischer Aufbau dieses Ansatzes wird in Abbildung 6 dargestellt.

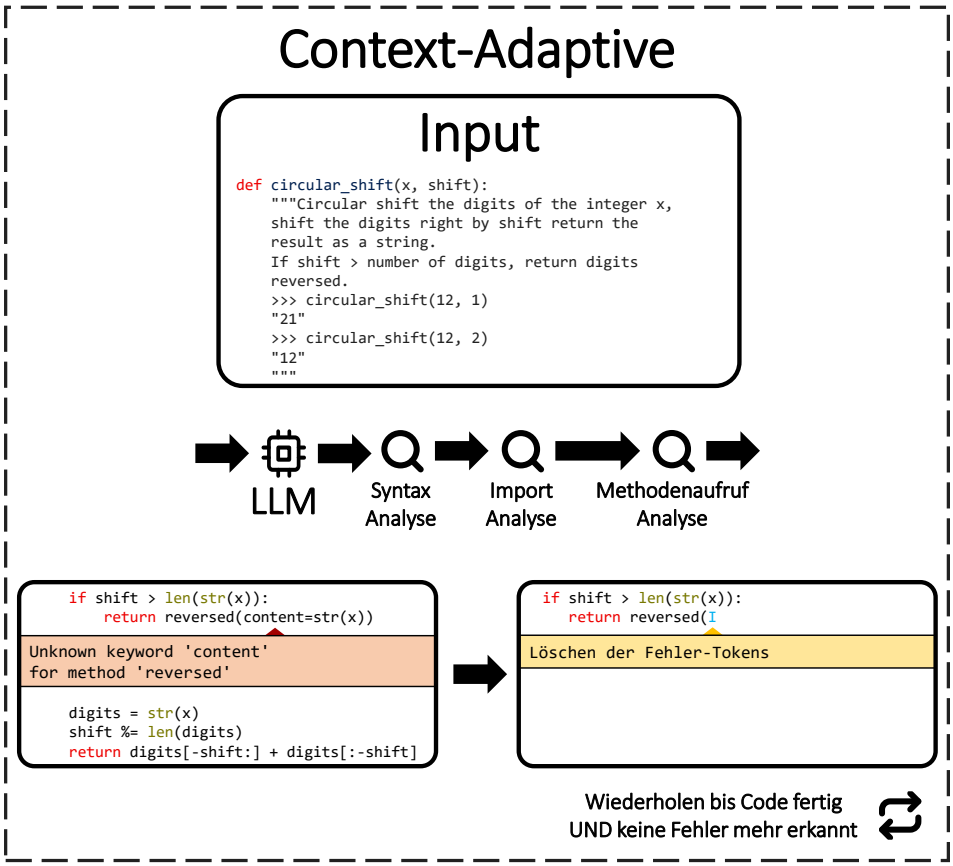


Abbildung 6: Verfeinfachte Darstellung des Context-Adaptive Samplers

3.6.2 Limitationen

Der vorgestellte Ansatz beschränkt sich auf eine Teilmenge von Fehlerquellen, die sich relativ einfach durch statische Prüfungen abfangen lassen. Dadurch werden jedoch weitere portenzielle Fehlerarten, die ebenfalls die Ausführbarkeit beeinträchtigen, nicht berücksichtigt und können ungehindert generiert werden.

Da sich dieser Ansatz ausschließlich auf die Verwendung statischer Analysetools stützt, können logische Fehler, die sich nur beim Ausführen des Programmcodes zeigen, nicht erkannt und behandelt werden.

4 Implementierungshinweise

In diesem Kapitel werden besondere Schwierigkeiten und Möglichkeiten bei der Implementation der zuvor vorgestellten Verfahren beleuchtet. Insbesondere werden Abweichungen zwischen der theoretisch konzipierten und der praktisch realisierten Funktionsweise herausgearbeitet.

4.1 Anbindung des Sprachmodells

Die eigentliche Generierung von Python-Code wird in allen vorgestellten Ansätzen über ein Sprachmodell realisiert, das innerhalb von Python aufgerufen wird. In der originalen Implementierung von RoCode erfolgt diese direkt durch das Python-Modul `transformers`[6] [31], was ein hohes Maß an Konfigurierbarkeit ermöglicht. Allerdings ist so das LM unmittelbar an die Code-Generierungsmethode gekoppelt und existiert somit ausschließlich innerhalb dieser Umgebung. Folglich greift dieser Ansatz nicht auf bereits vorhandene Infrastruktur zurück, die LMs zu Verfügung stellt. Würde einer der vorgestellten Ansätze zukünftig in ein verwendbares Produkt überführt, wäre eine separate Modellausführung alleinig für den Zweck der Codegenierierung erforderlich.

Zur Umgehung dieser Einschränkungen wird in der vorliegenden Arbeit das Sprachmodell als externer Service angebunden. Für diese Anbindung können verschiedene Softwarelösungen zum Einsatz kommen. Innerhalb des DLR hat sich die Software Ollama[7] als ein De-facto-Standard für die Ausführung von Sprachmodellen etabliert. Da die Schnittstelle von Ollama jedoch keine Möglichkeiten des Constraint-Decodings über das reine Generieren von JSON-Objekten hinaus bietet, ist dieser Service für die gegenwärtigen Anforderungen ungeeignet. Um das notwendige Maß an Flexibilität bei der Generierung zu gewährleisten, wird hier LlamaCPP eingesetzt, eine Software, auf die Ollama aufbaut [32].

4.2 Einfache Ansätze

Die Verfahren Temperature-Sampling, Sampling & Filtering sowie Post-Revising weisen vergleichsweise einfache Strukturen auf. Zu diesen Verfahren ist lediglich festzuhalten, dass

[6] `https://pypi.org/project/transformers/`
[7] `https://ollama.com/`

die unbegrenzte Fortsetzung der Codegenerierung bei auftretenden Fehlern in der Praxis durch Begrenzung der maximalen Anzahl von Versuchen limitiert wird. Daraus ergibt sich ein Abbruchmechanismus, der den Prozess beendet, wenn wiederholte Versuche kein zufriedenstellendes Ergebnis liefern. In der aktuellen Implementierung wird der Prozess nach zehn fehlerhaften Generationen abgebrochen, wobei die zuletzt generierte Variante als Vervollständigung zurückgegeben wird.

4.3 AdapT

Der AdapT-Ansatz unterteilt den Code in zwei Segmente: den Blockanfang, für den ein erhöhter Temperaturwert eingesetzt wird, sowie den übrigen Text, der mit niedriger Temperatur generiert wird. Folglich muss Code bis zum Beginn eines Blocks generiert werden. Ein Constraint, der sich jedoch nicht unmittelbar formal definieren lässt. In der praktischen Umsetzung generiert das Sprachmodell daher Segmente von jeweils zwanzig Tokens. Im Anschluss wird der daraus resultierende Teilcode mittels eines Python-Lexers in Code-Tokens transformiert. Dieses Verfahren ist nur möglich, wenn der generierte Text als Liste von Code-Tokens interpretiert werden kann. Bei Auftreten eines Fehlers wird dieser eliminiert und die Generierung wird ab der betreffenden Position fortgesetzt.

Anschließend erfolgt eine Analyse der Code-Tokens, um den Beginn eines neuen Blocks durch den Token `INDENT` zu identifizieren. Falls im Code ein neuer Abschnitt erkannt wird, erfolgt ab dessen Beginn die Generierung eines einzelnen Tokens mit angepasster Temperatur, gefolgt von einer fortgesetzten normalen Generierung.

Die praktische Umsetzung entspricht dem beschriebenen AdapT-Verfahren größtenteils. Jedoch erfordert die Praxis eine zusätzliche Übersetzungsprüfung des Textes in Code und die Erkennung der Codeblock-Anfänge erfolgt retrospektiv, was zu Rollbacks und einer erhöhten Gesamtanzahl an generierten Tokens führt.

4.4 RoCode

Für diesen Ansatz wurde die offizielle Implementierung von RoCode [31] herangezogen, die jedoch in geringem Umfang von dem im zugehörigen Paper beschriebenen Vorgehen (vgl. [5]) abweicht. Die Methodik beginnt damit, dass zwei Schranken definiert werden, die eine endlose Generierung verhindern. Es wird eine maximale Anzahl von insgesamt

generierten Tokens und eine maximale Länge des aktuell erzeugten Codes festgelegt.

Die statische Analyse des Codes erfolgt nicht zu jedem beliebigen Zeitpunkt, sondern erst nach jeder generierten Zeile. Zunächst wird der Code in einen AST überführt, wodurch die Syntax überprüft wird. Anschließend wird ein Ausführungstest durchgeführt. Falls der Code abgeschlossen ist, also die Generation vollständig beendet ist oder kein Syntaxfehler vorliegt und der Code mit vier aufeinander folgenden leeren Zeilen endet, wird die Methode mit verschiedenen Eingaben aufgerufen und die erhaltenen Ergebnisse verglichen. Falls seit den letzten Test kein neuer ausführbarer Code entstanden ist, wird die nächste Zeile generiert. Anderenfalls wird die Funktion mit denselben Eingaben gestartet, das Ergebnis jedoch ignoriert, um interne Funktionsfehler zu erkennen ohne auf vollständigen Code angewiesen zu sein. Für diesen Fall werden vor der Ausführung sämtliche *while*-Schleifen um ein *break* ergänzt, sodass der Code vollständig ausgeführt werden kann.

Sollte der Ausführungstest ein positives Ergebnis liefern, erfolgt eine Prüfung auf wiederholte Muster bestimmter Code-Segmente, was ebenfalls als Fehler gewertet wird. Wird auch hier kein Fehler festgestellt und der Code zuvor als abgeschlossen erkannt, wird er dem Nutzer zurückgegeben. Liegt die Position des Fehlers außerhalb des generierten Bereichs, wie es für eine falsche Methodenrückgabe zu erwarten ist, wird der Code bis zum Token mit der höchsten Entropie während der Generation zurückgerollt. Bei einem Syntaxfehler oder einem anderen Problem, das sich auf eine genaue Position zuordnen lässt, wird bis zum Beginn der betroffenen Codezeile zurückgeschrieben. Treten in derselben Zeile wiederholt Fehler auf, beginnt die Generierung vollständig neu. Wurde zuvor der Fehler als Musterwiederholung erkannt, so wird der Code ab dem Beginn der ersten Zeile des Musters gelöscht. In allen anderen Fehlerfällen wird zu dem Token mit der höchsten Entropie während der Generation zurückgerollt.

Wie sich aus dieser Vorgehensweise des praktischen Algorithmus erkennen lässt, unterscheidet sich die Methode von der zuvor vorgestellten Theorie. Ein Abbruchkriterium, das eine unendliche Schleife verhindert, wird eingefügt, und die statische Analyse um Ausführungstests ergänzt. Gleichzeitig werden je nach Art des Fehlers unterschiedliche Positionen als Anker für die Neugenerierung verwendet.

4.5 Context-Adaptive

Im Gegensatz zu RoCode werden die Prüfungen nicht zeilenweise, sondern nach jeweils zwanzig generierten Tokens durchgeführt. Da Tests somit auch mit unvollständigen Zeilen oder Code-Tokens möglich sind, erlaubt der implementierte Syntax-Check unvollständige Strukturen, sofern die vorhandenen Anfänge zu einem gültigen Muster erweitert werden können. Analog zu RoCode erfolgt die Umwandlung in einen AST mithilfe der Python-Bibliothek `lark`[8]. Im Anschluss wird der zuvor erstellte Syntaxbaum genutzt, um zu überprüfen, ob weitere Tests erforderlich sind. Hierbei werden die Positionen der relevanten Syntaxsegmente mit denen der zuletzt geprüften verglichen. Wenn die Codegenerierung durch das Modell abgeschlossen wurde, erfolgt eine weitere Überprüfung, ob der Gesamtcode valide ist, indem ein Code-Token eingesetzt wird, das an `lark` das Dateiende signalisiert. Zusätzlich wird der geschriebene Code dahingegen geprüft, ob er Inhalt hat oder lediglich eine leere Methodendeklaration darstellt.

Falls ein Test an einer beliebigen Stelle einen Fehler aufdeckt, werden die verbleibenden Tests übersprungen und stattdessen der Code ab der Fehlerposition sowie einige Tokens davor gelöscht. Falls der gleiche Code künftig erneut generiert wird, wird die Erzeugung des fehlerauslösenden Tokens verhindert.

4.5.1 Überprüfung der Imports

Die Inspiration dieser statischen Überprüfung entstammt der praktischen Python-Entwicklung. Viele für die Softwareentwicklung eingesetzte Editoren bieten eine Autovervollständigung an, die sämtliche im jeweiligen Kontext zulässigen Wörter umfasst. So werden beispielsweise nach Eingabe von „*import* “ installierte Bibliotheken vorgeschlagen.

Das Python-Modul `jedi`[9] stellt dieselben Funktionalitäten bereit, sodass diese Liste programmatisch abgerufen werden kann. Für jede *import*-Anweisung im Syntaxbaum des generierten Codes wird geprüft, ob das verwendete Modul in der Vorschlagsliste enthalten ist und ob sämtliche aus diesem importierten Klassen vorhanden sind.

Obwohl die Nutzung eines Sprachservers, wie `jedi`, für die Überprüfung von *import*-

[8] `https://github.com/lark-parser/lark`
[9] `https://github.com/davidhalter/jedi`

Anweisungen praktisch ist, weist dieser Ansatz ebenfalls einige Probleme auf. So werden Bibliotheken, die sich im lokalen Verzeichnis befinden und über einen lokalen Pfad eingebunden werden, nicht als Option vorgeschlagen und damit als falscher *import* markiert. Weiter führt auch das wiederholte Importieren von Klassen aus demselben Modul zu Problemen, die im Rahmen dieser Arbeit nicht behoben wurden.

4.5.2 Überprüfung von Methodenaufrufen

Auch hier wird auf die Vorschläge der Autovervollständigung zurückgegriffen. Diese lassen sich während eines Methodenaufrufs in drei Kategorien einordnen:

- **Keine Vorschläge**: Dies geschieht, wenn bereits sämtliche benötigten Argumente mit Schlüsselwörtern übergeben wurden. Wird an einer solchen Stelle ein weiteres Argument übergeben, so wird dieses in jedem Fall als Fehler markiert.
- **Nur Schlüsselwörter als Vorschläge**: Dies tritt ein, wenn die aufgerufene Methode keine Argumente ohne Schlüsselwörter akzeptiert oder bereits ein solches zuvor übergeben wurde. In diesem Fall werden alle folgenden Argumente ohne Schlüsselwort oder mit nicht in der Liste vorhandenen Schlüsselwort als Fehler markiert.
- **Kombination aus Schlüsselwörtern und Namen**: Dies gilt, solange die aufgerufene Methode noch Argumente ohne Schlüsselworte akzeptiert. In diesem Fall ist ein Argument nur dann fehlerhaft, wenn es ein Schlüsselwort verwendet, das nicht in dieser Liste enthalten ist.

Wie auch bei der Überprüfung von *import*-Anweisungen, bietet die Nutzung eines Sprachservers eine relativ einfache Möglichkeit, Fehler zu identifizieren, jedoch treten Probleme bei der Erkennung bestimmter Fälle auf. Besonders problematisch sind Methoden, die als Argument eine beliebige Liste von Schlüsselworten zulassen. Da jeder beliebige Name erlaubt ist, werden keine konkreten Schlüsselwörter als Vorschläge angezeigt, sodass jedes übergebene Argument mit Schlüsselwort als Fehler markiert wird. Zudem schlägt `jedi` Schlüsselwörter vor, selbst wenn eine Methode nur Argumente ohne Schlüsselworte akzeptiert. In solchen Fällen werden die Argumente daher falsch positiv ausgewertet.

5 Evaluierung

In diesem Kapitel werden die zuvor erklärten Generierungsmethoden anhand eines Test-Datensatzes für verschiedene Sprachmodelle bewertet. Zunächst wird der Versuchsaufbau dargelegt, anschließend werden die daraus gewonnenen Ergebnisse präsentiert.

5.1 Versuchsaufbau

Zur Sicherstellung einer Vergleichbarkeit der Resultate zwischen unterschiedlichen Studien wird der Datensatz HumanEval herangezogen: ein Benchmark, das in aktuellen Arbeiten als Referenzrahmen dient (vgl. [5, 6, 33, 34]).

HumanEval besteht aus 164 handgeschriebenen Programmieraufgaben und wurde von OpenAI entwickelt. Jede Aufgabe umfasst eine Funktionsdeklaration, eine Dokumentation, die den Zweck der Methode erläutert und Verwendungsbeispiele enthält, und mehrere Testfälle. Für die Generierung erhält ein Sprachmodell im Allgemeinen eine Anweisung zur Vervollständigung der nachfolgenden Funktion, den Methodenkopf, die Dokumentation und ggf. bereits generierte Teillösung. Die Testfälle werden zur Überprüfung verwendet.

Sofern durch eine Generierungsmethode nicht anders vorgegeben, werden für das Sampling die Standardwerte des LlamaCPP-Servers verwendet (vgl. [26]). Hierbei kommt eine Kombination verschiedener Sampling-Technologien zum Einsatz, darunter `Top-k` ($k = 40$), `Top-P` ($p = 0,95$) und `Temperature-Sampling` ($T = 0.80$).

Im Anschluss werden die Ergebnisse für verschiedene Sprachmodelle verglichen. Durch den Einsatz von Modellen in den Größen 7B, 3B, 1,5B und 0,5B der Qwen-2.5-Coder-Instruct Serie lässt sich der Einfluss der Modellgröße auf die Leistungsfähigkeit der Algorithmen untersuchen. Diese Modellreihe wurde ausgewählt, weil sie laut einer Meta-Studie die beste Performance im Vergleich zu anderen Modellen gleicher Größe erzielt [35]. Zur Ergänzung der Untersuchung des Unterschieds zwischen auf Codegenerierung ausgelegten Modellen und allgemein auf Anweisungen folgenden Modellen, werden auch Daten für das Ministral-3-3B-Instruct Modell erhoben.

5.2 Ergebnisse

Ein erster Überblick über die Leistungen der unterschiedlichen Generierungsmethoden und Sprachmodelle ergibt sich aus der sogenannten PassRate. Diese Kennzahl gibt an, welcher Prozentsatz der generierten Programme positive Testergebnisse erzielt hat, also fehlerfrei und logisch korrekt ist. In Tabelle 1 werden die PassRates für jeweils zwei Durchläufe des HumanEval-Datensatzes dargestellt. Hierbei werden die Methoden nach ihrem praktischen Aufbau in zwei Gruppen geteilt: feedbackfreie Verfahren, die nicht auf Rückmeldungen der Code-Ausführung angewiesen sind und lediglich statische Analysen durchführen, und ausführungsbasierte Algorithmen, die den generierten Code aktiv testen. Wie aus diesem Unterteilung zu erwarten, erzielen ausführungsbasierte Verfahren insgesamt bessere Resultate als die feedbackfreien Alternativen. Unter den ausführungsbasierten Methoden erreicht RoCode mit dem Modell Qwen2.5-Code-7B-Instruct eine PassRate von $95,4\%$, gefolgt von dem selben Verfahren mit dem 3B-großen Coder-Modell, das $93,9\%$ erreicht. Im Bereich der feedbackfreien Methoden erzielt der in dieser Arbeit entwickelte Context-Adaptive Ansatz mit Qwen2.5-Coder-3B-Instruct eine PassRate von $76,2\%$, gefolgt von AdapT mit demselben LM, das $71,3\%$ erreicht. Die vergleichsweise geringe Leistung des größten Modells bei den feedbackfreien Verfahren wird im Kapitel 6.1 genauer betrachtet.

Tabelle 1 verdeutlicht zudem, dass die Qualität mit steigender Modellgröße zunimmt,

Sprachmodelle	**Feedbackfreie Methoden**			**Ausführungsbasierte Methoden**		
	Temperature-Sampling	AdapT	Context-Adaptive	Sampling & Filtering	Post-Revising	RoCode
Ministral-3-3B-Instruct	46.6	53.4	51.8	81.1	89.9	86.9
Qwen2.5-Coder-0.5B-Instruct	17.1	16.5	39.6	48.8	30.8	57.0
Qwen2.5-Coder-1.5B-Instruct	39.0	39.9	61.9	82.9	61.6	86.9
Qwen2.5-Coder-3B-Instruct	61.9	71.3	**76.2**	92.1	81.4	93.9
Qwen2.5-Coder-7B-Instruct	49.4	46.3	55.8	89.9	89.6	**95.4**

Tabelle 1: PassRate (in %) verschiedener Sprachmodelle mit unterschiedlichen Generierungsmethoden auf dem HumanEval-Datensatz

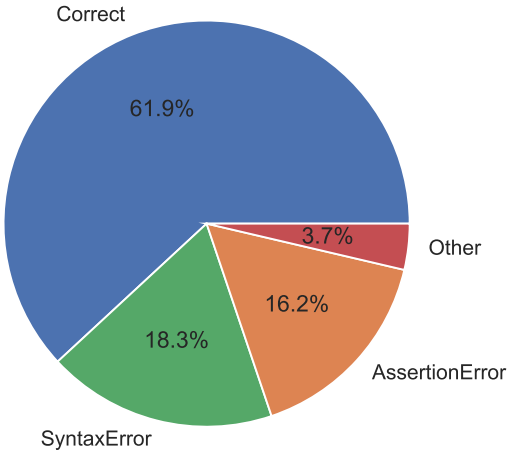


Abbildung 7: Fehler von Temperature-Sampling mit Qwen2.5-Coder-3B-Instruct

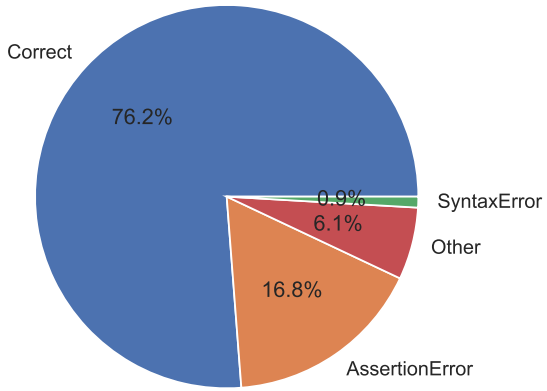


Abbildung 8: Fehler von Context-Adaptive mit Qwen2.5-Coder-3B-Instruct

ausgenommen die Anomalie von Qwen2.5-Coder-7B-Instruct, und dass das nicht speziell auf die Programmierung ausgelegte Modell Ministral-3-3B-Instruct in etwa mit dem 1,5B-Modell der Coding-Reihe vergleichbar ist.

Abbildung 7 zeigt die Fehlerverteilung des Temperature-Samplings mit Qwen2.5-Coder-3B-Instruct für die häufigsten Fehler. Der Anteil dieser Probleme wird in Abbildung 8 für den in dieser Arbeit entwickelten Kontext adaptiven Ansatz dargestellt. Hier lässt sich erkennen, dass der Anteil an syntaktischen Fehlern stark reduziert, der Anteil an korrekt generiertem Code sowie weitere, in diesem Abschnitt nicht näher beschriebenen, Fehlern erhöht wird. Der Anteil von logischen Fehlern, die falsche Rückgaben erzeugen und als AssertionError abgefangen werden, bleibt größtenteils unverändert.

Ein Vergleich des Token-Verbrauchs im Generierungsprozess zeigt, dass der Context-Adaptive Ansatz über sämtliche Größen des Coding-Modells hinweg den geringsten Verbrauch aufweist, während die Verfahren Sampling & Filtering und Post-Revising die höchsten Werte erzielen (vgl. Tabelle 2). Insgesamt schneiden die feedbackfreien Methoden in dieser Kennzahl besser ab, was vermutlich auf die geringeren Kontrollen zurückzuführen ist, die zu weniger erkannten Problemstellen führen.

Eine Analyse der für die Generierung benötigten Zeit, wie in Tabelle 3 dargestellt, folgt den Trends des Token-Verbrauchs. Anders als beim reinen Token-Verbrauch jedoch zeigen die Ergebnisse, dass feedbackfreien Methoden meist schneller sind als die ausführungsbasierten

Sprachmodelle	**Feedbackfreie Methoden**			**Ausführungsbasierte Methoden**		
	Temperature-Sampling	AdapT	Context-Adaptive	Sampling & Filtering	Post-Revising	RoCode
Ministral-3-3B-Instruct	54.0	81.8	61.9	269.6	286.6	350.5
Qwen2.5-Coder-0.5B-Instruct	1419.6	2187.8	70.1	8117.2	3701.1	878.4
Qwen2.5-Coder-1.5B-Instruct	475.9	612.0	67.3	3689.9	828.7	358.6
Qwen2.5-Coder-3B-Instruct	79.9	85.6	81.9	207.5	368.7	241.9
Qwen2.5-Coder-7B-Instruct	48.3	4**42.7**	45.3	**144.7**	387.2	182.3

Tabelle 2: Durchschnittlicher Token-Verbrauch der Codegenerierung verschiedener Sprachmodelle mit unterschiedlichen Generierungsmethoden auf dem HumanEval-Datensatz

Verfahren. Weiter ist die in dieser Arbeit etablierte Context-Adaptive Methode meist schneller als das reine Temperature-Sampling. RoCode hingegen ist trotz des geringeren Token-Verbrauchs im Vergleich zu den anderen ausführungsbasierten Verfahren der langsamste Algorithmus, über alle evaluierten Sprachmodelle hinweg.

Tabelle 4 zeigt die Verbesserung der einzelnen Algorithmen im Vergleich zum reinen Temperature-Sampling. Dabei verbessert Context-Adaptive die PassRate durchschnittlich

Sprachmodelle	**Feedbackfreie Methoden**			**Ausführungsbasierte Methoden**		
	Temperature-Sampling	AdapT	Context-Adaptive	Sampling & Filtering	Post-Revising	RoCode
Ministral-3-3B-Instruct	**0.6**	1.4	1.3	7.5	**6.9**	68.0
Qwen2.5-Coder-0.5B-Instruct	4.8	27.4	0.9	36.5	22.7	145.9
Qwen2.5-Coder-1.5B-Instruct	2.9	10.5	1.0	27.3	11.9	64.3
Qwen2.5-Coder-3B-Instruct	2.8	3.2	1.7	10.5	14.9	52.7
Qwen2.5-Coder-7B-Instruct	5.2	2.1	1.3	16.8	18.2	46.3

Tabelle 3: Durchschnittliche Dauer der Codegenerierung (in Sekunden) verschiedener Sprachmodelle mit unterschiedlichen Generierungsmethoden auf dem HumanEval-Datensatz

Sprachmodelle	**Feedbackfreie Methoden**			**Ausführungsbasierte Methoden**		
	Temperature-Sampling	AdapT	Context-Adaptive	Sampling & Filtering	Post-Revising	RoCode
Ministral-3-3B-Instruct	46.6	+14.6%	+11.2%	+74.0%	+92.9%	+86.5%
Qwen2.5-Coder-0.5B-Instruct	17,7	-6.8%	+123.7%	+175.7%	+74.0%	+222.0%
Qwen2.5-Coder-1.5B-Instruct	39.0	+2.3%	+58.7%	+112.6%	+57.9%	+122.8%
Qwen2.5-Coder-3B-Instruct	61.9	+15.2%	+23.1%	+48.8%	+31.5%	+51.7%
Qwen2.5-Coder-7B-Instruct	49.4	-6.3%	+13.0%	+82.0%	+81.4%	+93.1%

Tabelle 4: Relative PassRate-Verbesserung der Algorithmen im Vergleich zu reinem Temperature-Sampling

um $45,9\%$. Hiermit ist er effektiver als AdapT (durchschnittliche Verbesserung von $3,8\%$), jedoch schlechter als sämtliche ausführungsbasierten Methoden. Die größte Verbesserung erzielt RoCode, der durchschnittlich eine $115,2\%$ höhere PassRate als Temperature-Sampling erreicht.

6 Diskussion

Diese Kapitel untersucht und diskutiert die zuvor erhaltenen Ergebnisse kritisch. Dabei werden sowohl erwartete als auch unerwartete Verhaltensmuster betrachtet. Im Anschluss wird die in dieser Arbeit etablierte prototypische Methode einer qualitativen Analyse unterzogen und den Vergleichsalgorithmen gegenübergestellt; dadurch können die Trade-Offs zwischen Rechenzeit, Code-Qualität und Abhängigkeiten der Ansätze betrachtet werden. Anschließend wird die Forschungsfrage beantwortet und die gewonnenen Erkenntnisse zusammengefasst.

6.1 Leistung von Qwen2.5-Coder-7B-Instruct

Wie in Kapitel 5.2 festgestellt, sinkt die PassRate der Qwen-Modelle bei Anstieg von drei auf sieben Milliarden Parametern für feedbackfreie Methoden (vgl. Tabelle 1). Dies widerspricht dem allgemeinen Trend, dass größere Modelle tendenziell bessere Antworten liefern, und steht im Widerspruch zu einer erwarteten PassRate $88,4\%$ aus einer Meta-Studie [35].

Die Analyse der in Abbildung 9 dargestellten Ergebnisse, die das reine Temperature-

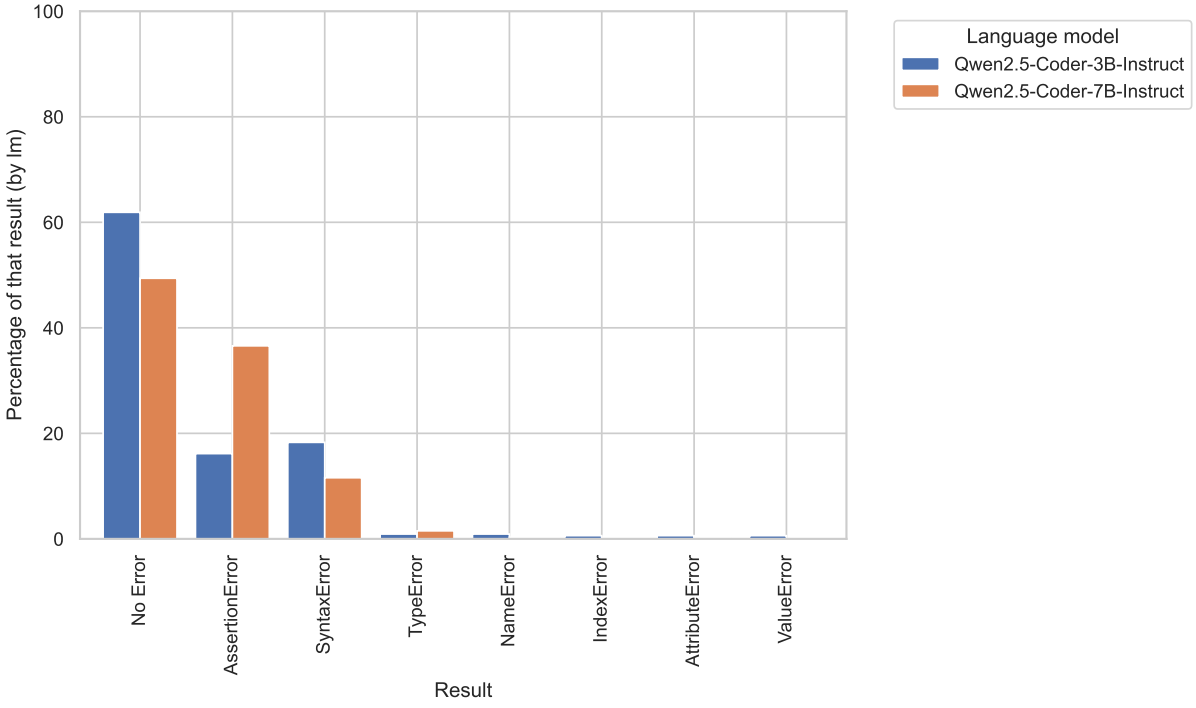


Abbildung 9: Vergleich von Qwen2.5-Coder-3B-Instruct und -7B-Instruct mit Temperature-Sampling

Sprachmodelle	Ministral-3 3B-Instruct	Qwen2.5-Coder 0.5B-Instruct	Qwen2.5-Coder 1.5B-Instruct	Qwen2.5-Coder 3B-Instruct	Qwen2.5-Coder 7B-Instruct
Leerer Code	25.0	31.6	20.3	17.0	86.7

Tabelle 5: Anteil (in %) von leeren Code unter Assertion-Fehlern verschiedener Sprachmodelle mit Temperature-Sampling auf dem HumanEval-Datensatz

Sampling der 3B- und 7B-Modell gegenüberstellt, zeigt, dass ein erheblicher Anteil der fehlenden Erfolgsquote auf Assertion-Fehler zurückzuführen ist. Eine detaillierte Untersuchung dieser fehlerhaften Generationen hat ergeben, dass 86, 7% dieser Fälle auf leeren Code zurückführen sind, während das kleinere 3B-Modell leeren Code nur lediglich in 17, 0% seiner Assertion-Fehler erzeugt (vgl. Tabelle 5). Unter „leeren Code“ versteht man hier Programmcode, der syntaktisch ausführbar ist, jedoch keine Funktion besitzt. Beispiele für dieses Verhalten sind in Quellcode 2 dargestellt.

Die Leistungsanomalie des Qwen2.5-Code-7B-Instruct-Modells lässt sich daher auf die Tendenz zurückführen, unter Verwendung des eingesetzten Prompt-Templates und Anweisungen leeren bzw. nicht funktionalen Code zu generieren.

```python
# Task: HumanEval/2
def truncate_number(number
    : float) -> float:
    """ Given a positive
        floating point
        number, it can be
        decomposed into
    and integer part (
        largest integer
        smaller than given
         number) and
        decimals
    (leftover part always
        smaller than 1).

    Return the decimal
        part of the number
        .
    >>> truncate_number
        (3.5)
    0.5
    """
    # Your code here
```

```python
# Task: HumanEval/25
def factorize(n: int) ->
    List[int]:
    """ Return list of
        prime factors of
        given integer in
        the order from
        smallest to
        largest.
    Each of the factors
        should be listed
        number of times
        corresponding to
        how many times it
        appeares in
        factorization.
    Input number should be
         equal to the
        product of all
        factors
    >>> factorize(8)
    [2, 2, 2]
    >>> factorize(25)
    [5, 5]
    >>> factorize(70)
    [2, 5, 7]
    """
    factors = []
    # Implement the
        function body here
    return factors
```

```python
# Task: HumanEval/82
def prime_length(string):
    """Write a function
        that takes a
        string and returns
         True if the
        string
    length is a prime
        number or False
        otherwise
    Examples
    prime_length('Hello')
        == True
    prime_length('abcdcba
        ') == True
    prime_length('kittens
        ') == True
    prime_length('orange')
         == False
    """
    pass
```

Quellcode 2: Beispielcode von Qwen-2.5-7B-Instruct mit Temperature-Sampling

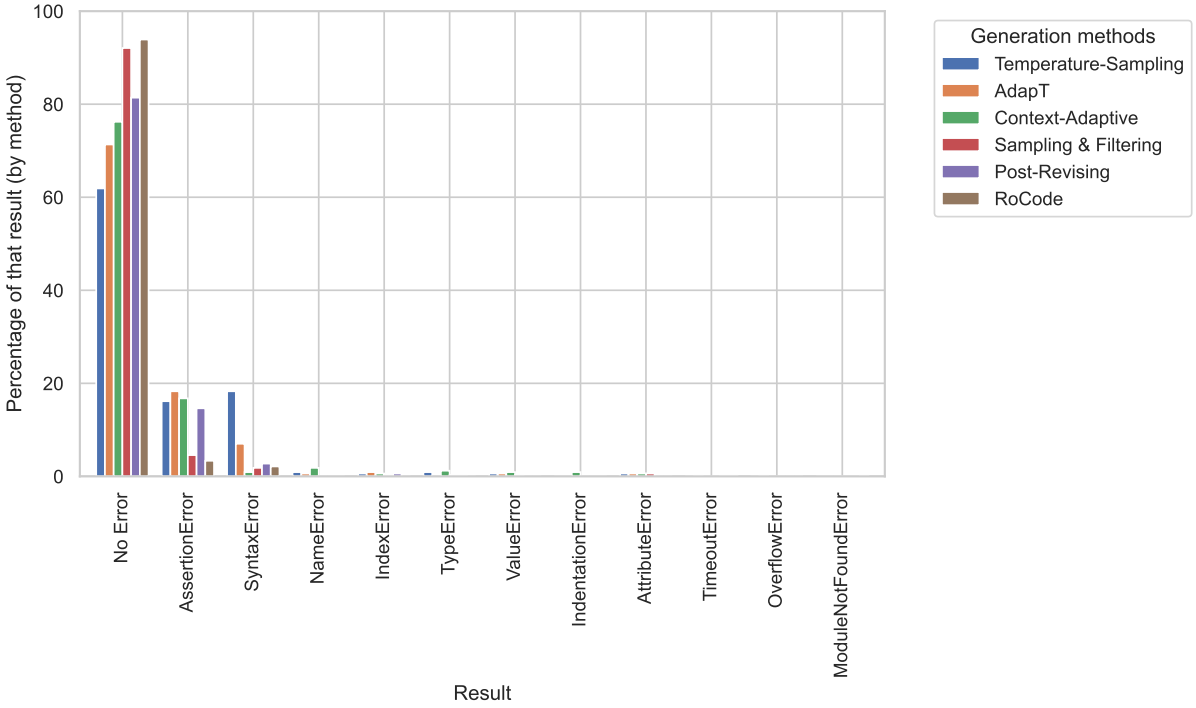


Abbildung 10: Vergleich der Generierungsmethoden mit Qwen2.5-Coder-3B-Instruct

6.2 Fehlervermeidung durch Kontext adaptiven Sampler

Abbildung 10 verdeutlicht die Verteilung von Fehlern, die, bei den in dieser Arbeit verglichenen Code-Generierungsmethoden, auf dem HumanEval-Datensatz auftreten. Der Context-Adaptive Sampler verfolgt in der aktuellen Implementierung zwei Hauptziele: die Vermeidung sämtlicher syntaktischen Fehler und die Verbesserung von Aufrufen an externe Bibliotheken. Letzteres lässt sich jedoch nicht unmittelbar anhand konkreter Fehlermeldungen quantifizieren und wird deshalb in einem späteren Kapitel detailliert analysiert.

Die Relevanz der syntaktischen Fehlervemeidung wird deutlich, wenn man beobachtet, dass $18,3\%$ der durch Temperature-Sampling generierten Programme aufgrund von Syntax-Fehlern nicht ausführbar sind. Im Vergleich dazu reduziert AdapT diese Rate auf $7,0\%$. Die in dieser Arbeit etablierte Methode erzielt eine Syntax-Fehlerrate von lediglich $0,9\%$, das entspricht drei fehlerhaften Ausgaben bei insgesamt 328 Testfällen.

Diese fehlerhaften Generationen werden in Quellcode 3 veranschaulicht. Alle drei Instanzen führen zu der Fehlermeldung: „Error: 'return' outside function“, da ein Rückgabe-Statement außerhalb eines Funktionskörpers steht. Obwohl dieser Fehler wie ein syntaktischer Fehler erscheint und Python einen SyntaxError ausgibt, ist eine solche Struktur

```
# Task: HumanEval/64
def vowels_count(s):
    """Write a function
        vowels_count which
         takes a string
        representing
    a word as input and
        returns the number
         of vowels in the
        string.
    [...]
    >>> vowels_count("
        ACEDY")
    3
    """
vowels = "aeiouAEIOU"
count = 0

for i in range(len(s)):
    if s[i] in vowels:
        count += 1
    elif s[i] == "y" or s[
        i] == "Y":
        if i == len(s) -
            1:
            count += 1

return count
```

```
# Task: HumanEval/101
def words_string(s):
    """
    You will be given a
        string of words
        separated by
        commas or spaces.
        Your task is
    to split the string
        into words and
        return an array of
         the words.

    For example:
    words_string("Hi, my
        name is John") ==
        ["Hi", "my", "name
        ", "is", "John"]
    words_string("One, two
        , three, four,
        five, six") == ["
        One", "two", "
        three", "four", "
        five", "six"]
    """
return s.split(',') + s.
    split(' ')
```

```
# Task: HumanEval/120
def maximum(arr, k):
    """
    Given an array arr of
        integers and a
        positive integer k
        , return a sorted
        list
    of length k with the
        maximum k numbers
        in arr.
    [...]
    Example 3:
        Input: arr = [-3,
            2, 1, 2, -1,
            -2, 1], k = 1
        Output: [2]
    [...]
    """
return sorted(arr)[-k:]
```

Quellcode 3: Fehlerhafter Code von Qwen-2.5-3B-Instruct mit Context-Adaptive Sampler

laut offizieller Python-Grammatik zulässig [30]. Somit sind die in Quellcode 3 dargestellten Programme, obwohl sie durch fehlende Einrückung des Funktionskörpers nicht korrekt ausgeführt werden können, nach der Definition dieser Arbeit syntaktisch korrekt (vgl. Kapitel 2.3.2). Daraus folgt, dass der hier entwickelte Ansatz tatsächlich sämtliche syntaktischen Probleme erfolgreich verhindert.

6.3 Effizienz von RoCode

Obwohl RoCode im Durchschnitt einen geringeren Token-Verbrauch aufweist als Sampling & Filtering und Post-Revising (vgl. Tabelle 2), ist dieser Ansatz unter den getesteten Verfahren die langsamste Generierungsmethode (vgl. Tabelle 3).

Die geringe Ausführungszeit von feedbackfreien Methoden lässt sich daraus ableiten, dass diese Ansätze keine Codeausführung durchführen und, sofern überhaupt, lediglich statische Analysen verwenden. Unter den ausführungsbasierten Verfahren verfügen Sampling & Filtering und Post-Revising über ein eingebautes Limit von zehn Versuchen, wodurch sie maximal zehnmal Code ausführen. Im Gegensatz dazu führt RoCode nach jeder generierten Codezeile, sofern kein Fehler durch ein statisches Analysewerkzeug festgestellt wurde, die generierten Zeilen aus: Dieser Ansatz erzeugt einen erheblichen Overhead.

Zusätzlich hat RoCode nach der Kontext adaptiven Methode die umfangreichste statische Analyselogik, was weitere Zeit während der Generierung beansprucht.

Insgesamt lässt sich daraus schließen, dass RoCode hinsichtlich der Anzahl der generierten Tokens vergleichsweise effizient ist, jedoch durch die komplexe Code-Analyse zu einem nicht besonders schnellen Algorithmus wird. Aufgrund des geringen Token-Verbrauchs im Vergleich zu einer PassRate von 95.4% (vgl. Tabelle 1) lohnt sich dieser Ansatz besonders dann, wenn externe Sprachmodelle wie Claude[10] oder ChatGPT[11] verwendet werden, die nach den verbrauchten Tokens abrechnen.

6.4 Qualitative Analyse des Context-Adaptive Samplers

Wie bereits in Kapitel 6.2 erwähnt, ist eine quantitative Erfassung von Fehlern bei der Verwendung von Schnittstellen und externen Programm-Bibliotheken schwierig. Um das Verhalten der verschiedenen Code-Generierungsmethoden, insbesondere des in dieser Arbeit konzipierten Ansatzes, systematisch zu vergleichen und zu analysieren, wurden daher gezielte Testfälle entwickelt.

Ein vergleichbares Problem wurde 2024 in einer Studie behandelt, in der eine Sprachmodell speziell für die kohärente Nutzung von Programm-Schnittstellen optimiert wurde (vgl. [7]). Das daraus resultierende Gorilla-Modell kombiniert Fine-Tuning mit dem Zugriff auf die Dokumentation von Schnittstellen, um korrekte Aufrufe zu generieren [7].

Aus dieser Arbeit ist das Berkeley Function Calling Leaderboard (BFCL) entstanden, das auf einem speziell zugeschnittenen Testdatensätze basiert. Eine Aufgabe besteht in diesem Rahmen aus einem in Klartext formulierten Problem und einer Liste von Funktionen, die zur Lösung herangezogen werden können [36].

Da sich das Format dieser Aufgaben grundlegend von dem des HumanEval-Datensatzes[12] unterscheidet, können die Testfälle nicht direkt übernommen werden. Zudem ist der in dieser Arbeit verwendete Testaufbau nicht für die Übergabe von relevanten Funktionen und Klartext-Aufgaben ausgelegt. Unter Inspiration der BFCL-Testfälle wurden daher fünf eigene Tests und ein weiterer Testfall formuliert, die nun näher betrachtet werden.

[10] `https://claude.ai/login`
[11] `https://chatgpt.com/`
[12] `https://github.com/openai/human-eval/tree/master/data`

```python
# Task: BFCL_v3_simple_3
def solve_quadratic(a : List[float], r : float) -> Tuple[float, float]:
    """
    Returns the two solutions to the quadratic: a[0]x^2 + a[1]x + a[2] = r as a Tuple
    This function utilizes 'quadratic_roots' from the 'custom' module.

    >>> solve_quadratic([4, 12, -9], -2)
    (-3.5, 0.5)
    >>> solve_quadratic([4, -8, 4], 4)
    (0, 2)
    >>> solve_quadratic([-13, -2, 5], 2.8)
    (0.34158, -0.49543)
    """
```

Quellcode 4: Aufgabe zur Lösung quadratischer Gleichungen

Lösen quadratischer Gleichungen In dieser von `BFCL_v3_simple_3` (vgl. [36]) inspirierten Aufgabe soll eine Methode implementiert werden, die eine quadratische Gleichung löst. Hierfür wird eine Hilfsfunktion bereitgestellt, die die Nullstellen einer Parabel bestimmen kann (siehe Quellcode 4). Ziel dieses Tests ist die Überprüfung, ob die verwendeten Sprachmodelle die benötigte Methode eines bestimmten Moduls korrekt einbinden können.

Unter den untersuchten Methoden haben ausschließlich AdapT und der prototypische Kontext adaptive Ansatz fehlerhaften Code erzeugt (siehe Anhang A). AdapT importiert die benötigte Methode nicht, während Context-Adaptive bei einer eigenen Implementierung einer quadratischen Lösungsgleichung undefinierte Variablen verwendete, obwohl die Hilfsmethode korrekt eingebunden wurde. Die Nicht-Verwendung der importierten Funktion lässt sich entweder durch stochastische Zufälligkeit erklären, da weiterhin aus den Ausgabetokens gesampelt wird, oder durch eine inkorrekte Verwendung der Methode bzw. ein falsch negatives Ergebnis im integrierten Analyseverfahren.

Berechnung der kumulativen Wahrscheinlichkeit Die nächste Aufgabe prüft die Reaktion von Sprachmodellen auf unbekannte Schnittstellen. Es soll eine Methode zur Berechnung der Wahrscheinlichkeit für bis zu `max_heads`-mal Kopf bei `tosses` Münzwürfen implementiert werden (vgl. `BFCL_v3_simple_115` in [36]). Für diese Aufgabe steht die Hilfsfunktion `calculate_binomial_propability` bereit. Da den Modellen jedoch keine Informationen über das genaue Format oder die benötigten Parameter vorliegen, müssen sie den Kontext allein aus dem Methodennamen ableiten. `calculate_binomial_propability` erwartet drei Parameter k, n und p und berechnet die Wahrscheinlichkeit für das exakt k-malige Eintreten eines Ereignisses mit Wahrscheinlichkeit p bei insgesamt n Versuchen.

Keiner der untersuchten Ansätze konnte diese Aufgabe vollständig erfüllen. Ausschließlich RoCode rief die Hilfsfunktion mit korrekten Parametern auf. Die Ergebnisse der übrigen Methoden unterscheiden sich kaum und deuten darauf hin, dass bei der Generierung dieser Implementierung die Notwendigkeit der Kumulativität nicht erkannt wurde (siehe Anhang B). Die inkorrekten Übergaben von Werten an die Hilfsfunktion beim Context-Adaptive Sampler verdeutlicht, dass eine reine Überprüfung von übergebenen Schlüsselwörtern für die Bewertung von Methodenaufrufen nicht ausreichend ist. Das statische Analysetool weist damit gravierende Schwachstellen auf.

Berechnung von steigenden und sinkenden Aktienkursen Diese Aufgabe, angelehnt an `BFCL_v3_simple_144` (vgl. [36]), untersucht, wie verschiedene Generierungsmethoden mit inkorrekten Informationen über den Zweck von Methoden umgehen. Hierfür soll die relative Verbesserung des Werts einer Aktie in einem bestimmten Jahreszeitraum im Vergleich zum Startwert ermittelt werden. Im Methodentext wird die Veränderung fälschlicherweise als prozentuale Steigung beschrieben. Damit werden Modelle angeregt, Werte in Prozent auszugeben.

Nur Sampling & Filtering schafft es, die Aufgabe mit korrekten Werten abzuschließen. Temperature-Sampling, Post-Revising und RoCode gaben Werte in Prozent zurück. AdapT strebt ebenfalls die Rückgabe prozentualer Werte an, importiert jedoch nicht die benötigte Hilfsmethode. Der Kontext adaptive Ansatz erzeugt als einziger einen inkorrekten Methodenaufruf an die Hilfsfunktion und enthielt zusätzlich fehlerhafte Logik (siehe Anhang C). Diese Ergebnisse unterstreichen erneut die unzureichende Überprüfung von Methodenaufrufen durch diesen Ansatz. Bei Parametern ohne explizite Schlüsselwörtern greift das Verfahren nur bedingt ein. Aus den Antworten der ausführungsbasierten Methoden lässt sich zudem schließen, dass die Bereitstellung von Testfällen und die Ausführung von generiertem Code nicht zwangsläufig zur einer Verbesserung der menschlich verursachten Fehler führt.

Berechnung des Kreisumfangs In diesem, von `BFCL_v3_parallel_65` (vgl. [36]) inspirierten Test, soll die Verwendung von nicht-intuitiven Schnittstellen überprüft werden. Für eine Reihe von Radien muss der Umfang der entsprechenden Kreise berechnet werden. Hierfür steht eine Hilfsfunktion zur Verfügung, die bei Übergabe des Durchmessers den

Umfang bestimmt. Da die Funktion lediglich `calculate_circumference` heißt, tendieren die Generierungsmethoden dazu, den bereitgestellten Radius direkt zu übergeben.

Temperature-Sampling, Sampling & Filtering und RoCode setzten eigene Methoden zur Umfangsberechnung ein, ohne auf die Hilfsfunktion zurückzugreifen. Diese Ansätze sind der Schwierigkeit dieser Aufgabe somit ausgewichen und lieferten korrekt Ergebnisse. AdapT und Post-Revising verwendeten die externe Methode, übergaben jedoch den Radius als Parameter, wodurch eine falsche Ausgabe entstand. Der Kontext adaptive Ansatz zeigt dasselbe Verhalten, wobei dieser die benötigte Methode nicht importiert und ebenfalls den Radius übergibt (siehe Anhang D). Die wiederholte Verwendung von nicht geladenen Methoden verdeutlicht die Notwendigkeit, die Analyse von Methodenaufrufen um die Prüfung auf die Existenz der Methode zu erweitern.

Identifizierung des größten Kreises Diese Aufgabe, angelehnt an `BFCL_v2_parallel_67` (vgl. [36]), prüft die Fähigkeit der Sprachmodelle, einen geplanten Funktionsablauf zu optimieren. Ziel ist es, aus einer Liste von Radien den Kreis mit der größten Fläche zu identifizieren. Für die Flächenberechnung steht eine Hilfsfunktion bereit, die den Radius annimmt. Da der Kreis mit dem größten Radius die größte Fläche besitzt, ist diese Aufgabe auch ohne Aufruf der Hilfsfunktion lösbar.

Alle Generierungsmethoden, mit Ausnahme von Temperature-Sampling, lieferten korrekte Ausgaben, wobei keine der Methoden die Logik optimiert hat. Temperature-Sampling nutzt zwar ebenfalls die Hilfsfunktion, importiert diese jedoch nicht, was zu einer fehlerhaften Ausführung führt. Auch der Kontext adaptive Ansatz hat die Hilfsfunktion nicht geladen und ersetzte sie durch eine eigene Implementierung, die dieselbe Funktionalität bereitstellt (siehe Anhang E). Aus dieser Aufgabe lässt sich ableiten, dass keine der in dieser Arbeit vorgestellten Generierungsmethoden die Optimierung des generierten Codes unterstützt.

Praxistest Ein abschließender Test, der auf internem Code des DLR und den gesammelten Erfahrungen bei der autonomen Codegenerierung basiert, untersucht die Verwendung von Schnittstellen, bei denen die Übergabe von Schlüsselwerten üblich ist. Hierfür soll eine Verbindung zum Datenbankservice ArangoDB[13] deklariert werden, wobei einer Me-

[13] `https://arango.ai/`

thode eine URL und Portnummer übergeben werden. Da allerdings der Python ArangoDB Driver keinen Port als separates Argument akzeptiert, muss die gesamte Netzadresse, einschließlich des Ports, als Host übergeben werden [37].

Diese Aufgabe wurde von Context-Adaptive, Sampling & Filtering und RoCode korrekt bearbeitet, während Temperature-Sampling, AdapT und Post-Revising versuchten, Port-Argumente einzusenden (siehe Anhang F). Daraus lässt sich schließen, dass der Kontext adaptive Ansatz die Überprüfung von Schlüsselargumenten zuverlässig durchführt. Ein ähnlicher Effekt kann jedoch auch durch die Ausführung von generiertem Code erreicht werden.

Insgesamt zeigen die sechs Testfälle, dass keiner der untersuchten Algorithmen sämtliche Aufgaben erfolgreich abschließen konnte. Zwar funktioniert weiter die Analyse der Funktionsaufrufe im aktuellen Context-Adaptive Sampler, jedoch ist sie nicht ausreichend umfassend, um für allgemeine Aufgaben signifikante Verbesserungen zu erzielen.

6.5 Weitere Fähigkeiten der Generierungsmethoden

In Bezug auf die autonome Softwareentwicklung durch Sprachmodelle ist von besonderem Interesse, ob die Generierungsprozesse eine Fehlerfreiheit des erstellten Codes garantieren können. Offensichtlich ist dies bei reinem Temperature-Sampling nicht der Fall, da diese Methode zufällige Tokens ohne weitere Überprüfung auswählt. Auch der AdapT Ansatz kann keine Garantien bezüglich von Fehlern geben, da in dem theoretischen Vorgehen keine Analysen des Codes vorgesehen ist und weiterhin Tokens entsprechend einer Wahrscheinlichkeitsverteilung ausgewählt werden. In der praktischen Implementierung von AdapT wird hingegen eine Analyse eingebaut, die den Eingabetext zur Identifikation von Codeblöcken in Quellcode übersetzt. Dadurch kann gewährleistet werden, dass der generierte Code aus Quellcode-Wörtern besteht; die Fehlerfreiheit des Syntax wird jedoch nicht betrachtet.

Anders hierzu gewährleisten Sampling & Filtering und Post-Revising eine syntaktische und logische Korrektheit, sofern ausreichend Testfälle deklariert sind. in der Praxis ist jedoch eine maximale Anzahl an Versuchen integriert, sodass diese Garantie nur theoretisch besteht.

Auch RoCode bietet theoretisch eine Garantie der Fehlerfreiheit im Hinblick auf die in-

tegrierten statischen Analysen, da solange Fehler vorliegen, diese entfernt und der entsprechende Code mit abgewandelten Wahrscheinlichkeiten neu generiert wird. Für den praktischen Algorithmus wurde jedoch ein maximales Generationslimit eingeführt, wodurch die Ausgabe eventuell bereits entdeckte Fehler enthalten kann. Da RoCode in der Praxis durch die Ausführung des Programmcodes erweitert wird, gewinnt die interne Fehleranalyse an Tiefe und kann sämtliche logische Probleme behandeln.

Der in dieser Arbeit etablierte Context-Adaptive Sampler verwendet dezidierte statische Analysen, um bestimmte Code-Abschnitte zu überwachen. Dadurch entsteht erneut eine allgemeine Garantie der Fehlerfreiheit hinsichtlich der integrierten Tests. Da die statischen Werkzeuge weiter auf einer Syntax-Analyse beruhen, garantiert diese Methode die korrekte Syntax in der Ausgabe (vgl. Kapitel 6.2). Über die Fehlerfreiheit hinsichtlich der Verwendung externer Programm-Bibliotheken lässt sich aufgrund der unzureichenden statischen Analyse der verwendeten Tools (vgl. Kapitel 6.4 keine Aussage treffen.

Eine weitere Stärke des Kontext adaptiven Ansatzes ist die effiziente Erweiterbarkeit der Analyseumgebung. Anders als bei RoCode integriert diese Methode einen Mechanismus, der Werkzeuge nur bei Bedarf aktiviert und so auch bei einer Vielzahl von Tools eine hohe Effizienz ermöglicht. Darüber hinaus zeigte die verwendete syntaktische Analyse, dass durch gezielte Anwendung dieser Tests bestimmte Fehler vollständig unterdrückt werden können. Dadurch kann der Ansatz progressiv verbessert und an spezifische Bedingungen angepasst werden.

6.6 Fazit

Die vorliegende Arbeit hat gezeigt, dass die alleinige Anwendung von Temperature-Sampling zur autonomen Lösung von Programmieraufgaben durch Sprachmodelle zu einer Vielzahl unterschiedlicher Fehler führt. Um diese zu unterdrücken, wurden mehrere etablierte und moderne Algorithmen beschrieben und implementiert. Zusätzlich wurde ein weiteres Verfahren konzipiert, das sich durch einen modularen Aufbau und ausschließlich statische Analysen auszeichnet.

Auf dem HumanEval-Datensatz demonstriert der RoCode-Ansatz die beste PassRate. Unter den Methoden, die nicht auf die Ausführung des generierten Programms angewiesen sind, erweist sich der neu entwickelte Kontext adaptive Ansatz als Spitzenreiter.

Eine detaillierte Analyse der Resultate zeigt, dass der Context-Adaptive Sampler syntaktische Fehler vollständig vermeidet, was ihn von den übrigen Verfahren in der Praxis abhebt. Gleichzeitig wurden Lücken in der derzeit verwendeten statischen Analyse für die Überprüfung bei der Verwendung von externen Bibliotheken aufgezeigt. In diesem Bereich konnte der prototypische Ansatz keine Verbesserung erzielen. Insgesamt empfiehlt sich RoCode insbesondere bei der Nutzung von Sprachmodellen, die nach generierten Tokens abrechenen, da er das beste Verhältnis zwischen Token-Verbrauch und Antwortqualität aufweist. Sind allerdings nicht ausreichend Testfälle vorhanden oder sollte eine schnelle Antwort erforderlich sein, ist der Context-Adaptive Sampler die beste Wahl.

6.7 Beantwortung der Forschungsfrage

Die Forschungsfrage lautete:

> Können Methoden zur Syntax- und API-gesteuerten Beschränkung bei der Token-Generierung die Fehlerrate von erzeugtem Python-Code signifikant reduzieren?

Sie lässt sich folgendermaßen beantworten:

Durch eine in dieser Arbeit etablierte Generierungsmethode ist es möglich, syntaktisch korrekten Python-Code zu erzeugen. Ein signifikanter Einfluss auf die Fehlerhäufigkeit bei der Verwendung von Schnittstellen konnte hingegen nicht erzielt werden.

7 Ausblick

Der in dieser Arbeit prototypisierte, Kontext adaptive Generierungsansatz weist bereits mehrere vielversprechende Eigenschaften auf. Er ist sowohl im Token-Verbrauch als auch in der Generierungszeit sehr effizient. Darüber hinaus erfordert die Methode keine zuvor bekannten Tests oder die Ausführung des generierten Codes, was eine leichte Integration in bestehende Developer-Pipelines ermöglicht. Durch den modularen Aufbau der statischen Analysetools lassen sich gezielt verschiedene Fehlerquellen ansteuern, während die Effizienz durch die Aktivierung ausschließlich notwendiger Tests weitere gewährleistet bleibt.

Potenzielle Weiterentwicklungen

- **Generalisierung des Algorithmus** Der vorgestellte Ansatz könnte dahingegen erweitert werden, dass er beliebige Programmieranfragen in beliebigen Format verarbeiten und auf existierenden Code aufbauen kann. Dadurch würde diese Methode zu einer Zwischenschicht zwischen dem Anwender und einem Sprachmodell werden, die direkt in jeden Entwicklungsprozess integriert werden kann.
- **Verbesserung der statischen Analyse** Die derzeit verwendete statische Analyse zur Prüfung der Verwendung externer Bibliotheken kann verbessert werden, um die Kohärenz des generierten Codes zu erhöhen. Ergänzend könnten weitere Analyse-Tools implementiert werden, die zusätzliche Fehlerquellen adressieren und damit die Gesamtqualität des Generierungsprozesses steigern.
- **Erweiterung des Algorithmus** Das interne Constraint-Decoding des Ansatzes könnte um einen RollBack-Mechanismus ähnlich dem von RoCode ergänzt werden, um bereits frühzeitig alternative Code-Stücke für fehlerhafte Ausgaben zu generieren. Weitere Aspekte, wie die adaptive Temperatur von AdapT oder das RollBack auf Tokens mit hoher Entropie bei RoCode, sollten hinsichtlich ihres Einflusses auf den Context-Adaptive Algorithmus untersucht werden.

Durch die Umsetzung dieser Ideen kann die Methode zukünftig noch robuster, flexibler und leistungsstärker werden, was sie zu einer wertvollen Komponente moderner Software-Entwicklungs-Workflows machen kann.

8 Literaturverzeichnis

[1] GitHub, "Octoverse: A new developer joins github every second as AI leads TypeScript to become the most popular language." `https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/`, October 2025. Zugriff am: 16.07.2026.

[2] Z. Zhang, C. Wang, Y. Wang, E. Shi, Y. Ma, W. Zhong, J. Chen, M. Mao, and Z. Zheng, "Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation," *Proc. ACM Softw. Eng.*, vol. 2, June 2025.

[3] M. Mahade Hasan, M. Waseem, K.-K. Kemell, J. Rasku, J. Ala-Rantala, and P. Abrahamsson, "Assessing small language models for code generation: An empirical study with benchmarks," *Journal of Systems and Software*, vol. 236, p. 112815, 2026.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[5] X. Jiang, Y. Dong, Y. Tao, H. Liu, Z. Jin, and G. Li, "Rocode: Integrating backtracking mechanism and program analysis in large language models for code generation," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 334–346, 2025.

[6] Y. Zhu, J. Li, G. Li, Y. Zhao, J. Li, Z. Jin, and H. Mei, "Hot or cold? adaptive temperature sampling for code generation with large language models," in *Proceedings of the Thirty-Eighth AAAI Conference on Artificial Intelligence and*

Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence and Fourteenth Symposium on Educational Advances in Artificial Intelligence, AAAI'24/IAAI'24/EAAI'24, AAAI Press, 2024.

[7] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," in *Advances in Neural Information Processing Systems* (A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, eds.), vol. 37, pp. 126544–126565, Curran Associates, Inc., 2024.

[8] R. Bommasani, P. Liang, and T. Lee, "Holistic evaluation of language models," *Annals of the New York Academy of Sciences*, vol. 1525, no. 1, pp. 140–146, 2023.

[9] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (A. Moschitti, B. Pang, and W. Daelemans, eds.), (Doha, Qatar), pp. 1724–1734, Association for Computational Linguistics, Oct. 2014.

[10] K. Muludi, K. M. Fitria, J. Triloka, and Sutedi, "Retrieval-augmented generation approach: Document question answering using large language model," *International Journal of Advanced Computer Science and Applications*, vol. 15, no. 3, 2024.

[11] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (K. Erk and N. A. Smith, eds.), (Berlin, Germany), pp. 1715–1725, Association for Computational Linguistics, Aug. 2016.

[12] T. Kudo and J. Richardson, "SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (E. Blanco and W. Lu, eds.), (Brussels, Belgium), pp. 66–71, Association for Computational Linguistics, Nov. 2018.

[13] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations," in *Proceedings of the 2013 Conference of the North American*

Chapter of the Association for Computational Linguistics: Human Language Technologies (L. Vanderwende, H. Daumé III, and K. Kirchhoff, eds.), (Atlanta, Georgia), pp. 746–751, Association for Computational Linguistics, June 2013.

[14] M. Shao, A. Basit, R. Karri, and M. Shafique, "Survey of different large language model architectures: Trends, benchmarks, and challenges," *IEEE Access*, vol. 12, pp. 188664–188706, 2024.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, (Red Hook, NY, USA), p. 6000–6010, Curran Associates Inc., 2017.

[16] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," tech. rep., OpenAI, 2018.

[17] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation," in *Proceedings of the First Workshop on Neural Machine Translation* (T. Luong, A. Birch, G. Neubig, and A. Finch, eds.), (Vancouver), pp. 56–60, Association for Computational Linguistics, Aug. 2017.

[18] X. Yang, J. Yu, J. Wu, S. Sun, J. Wang, and Y. Yang, "Internalize the temperature: On-policy self-distillation as policy reheater for reinforcement learning," 2026.

[19] A. Fan, M. Lewis, and Y. Dauphin, "Hierarchical neural story generation," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (I. Gurevych and Y. Miyao, eds.), (Melbourne, Australia), pp. 889–898, Association for Computational Linguistics, July 2018.

[20] J. Hewitt, C. D. Manning, and P. Liang, "Truncation sampling as language model desmoothing," 2022.

[21] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," 2020.

[22] S. Geng, M. Josifoski, M. Peyrard, and R. West, "Grammar-constrained decoding for structured NLP tasks without finetuning," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing* (H. Bouamor,

J. Pino, and K. Bali, eds.), (Singapore), pp. 10932–10952, Association for Computational Linguistics, Dec. 2023.

[23] F. Raspanti, T. Ozcelebi, and M. Holenderski, "Grammar-constrained decoding makes large language models better logical parsers," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)* (G. Rehm and Y. Li, eds.), (Vienna, Austria), pp. 485–499, Association for Computational Linguistics, July 2025.

[24] ggml-org contributors, "llama.cpp." `https://github.com/ggml-org/llama.cpp`, 2026. GitHub repository, abgerufen am 23. Juli 2026.

[25] Ollama, "Ollama api reference." `https://docs.ollama.com/api`, 2026. Offizielle API-Dokumentation, abgerufen am 23. Juli 2026.

[26] ggml-org contributors, "llama.cpp http server documentation." `https://github.com/ggml-org/llama.cpp/blob/master/tools/server/README.md`, 2026. GitHub repository, abgerufen am 23. Juli 2026.

[27] W. T. Fitch and A. D. Friederici, "Artificial grammar learning meets formal language theory: an overview," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 367, pp. 1933–1955, 07 2012.

[28] N. Chomsky, "Three models for the description of language," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 113–124, 1956.

[29] N. Chomsky, "On certain formal properties of grammars," *Information and Control*, vol. 2, no. 2, pp. 137–167, 1959.

[30] Python Software Foundation, "Cpython peg grammar specification (python.gram)." `https://github.com/python/cpython/blob/main/Grammar/python.gram`, 2026. Datei aus dem offiziellen CPython-Repository, abgerufen am 23. Juli 2026.

[31] X. Jiang, Y. Dong, Y. Tao, H. Liu, Z. Jin, W. Jiao, and G. Li, "Rocode: Integrating backtracking mechanism and program analysis in large language models for code generation." `https://github.com/jiangxxxue/ROCODE`, 2026. GitHub repository, abgerufen am 30. Juli 2026.

[32] llama-cpp.com, “Llama.cpp vs ollama – which local llm tool is better?.” https://llama-cpp.com/llama-cpp-vs-ollama/. Webartikel, abgerufen am 30. Juli 2026.

[33] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, “Codet5+: Open code large language models for code understanding and generation,” 2023.

[34] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, “Wizardcoder: Empowering code large language models with evol-instruct,” 2024. Cited by: 133.

[35] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A survey on large language models for code generation,” *ACM Trans. Softw. Eng. Methodol.*, vol. 35, Jan. 2026.

[36] F. Yan, H. Mao, C. C.-J. Ji, T. Zhang, S. G. Patil, I. Stoica, and J. E. Gonzalez, “Berkeley function calling leaderboard.” https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html, 2024.

[37] Arango, “Arangodb python driver.” https://docs.arango.ai/ecosystem/drivers/python/, 2026. Dokumentation des ArangoDB Python driver, abgerufen am 10. August 2026.

A Generierte Antworten für BFCL_v3_simple_3

```python
# Task: BFCL_v3_simple_3 (Temperature-Sampling)
def solve_quadratic(a : List[float], r : float) -> Tuple[float, float]:
    """
    Returns the two solutions to the quadratic: a[0]x^2 + a[1]x + a[2] = r as a Tuple
    This function utilizes 'quadratic_roots' from the 'custom' module.

    >>> solve_quadratic([4, 12, -9], -2)
    (-3.5, 0.5)
    >>> solve_quadratic([4, -8, 4], 4)
    (0, 2)
    >>> solve_quadratic([-13, -2, 5], 2.8)
    (0.34158, -0.49543)
    """
    import custom
    return custom.quadratic_roots(a[0], a[1], a[2] - r)

# Task: BFCL_v3_simple_3 (AdapT)
def solve_quadratic(a : List[float], r : float) -> Tuple[float, float]:
    """[...]"""
    return quadratic_roots(a[0], a[1], a[2], r)

# Task: BFCL_v3_simple_3 (Context-Adaptive Sampler)
def solve_quadratic(a : List[float], r : float) -> Tuple[float, float]:
    """[...]"""
    # Import the function from the custom module
    from custom import quadratic_roots

    # Calculate the discriminant
    discriminant = b**2 - 4*a*c

    # Solve for x using quadratic formula
    if discriminant > 0:
        root1 = (-b + math.sqrt(discriminant)) / (2*a)
        root2 = (-b - math.sqrt(discriminant)) / (2*a)
    elif discriminant == 0:
        root1 = root2 = -b / (2*a)
    else:
        root1 = root2 = complex(-b / (2*a), math.sqrt(-discriminant) / (2*a))

    # Check if the roots satisfy the quadratic equation
    if a[0]*root1**2 + a[1]*root1 + a[2] != r:
        raise ValueError("One or both roots do not satisfy the quadratic equation.")
    if a[0]*root2**2 + a[1]*root2 + a[2] != r:
        raise ValueError("One or both roots do not satisfy the quadratic equation.")

    return (root1, root2)
```

```python
# Task: BFCL_v3_simple_3 (Sampling & Filtering)
def solve_quadratic(a : List[float], r : float) -> Tuple[float, float]:
    """[...]"""
    from custom import quadratic_roots
    x1, x2 = quadratic_roots(a[0], a[1], a[2] - r)
    return (x1, x2)

# Task: BFCL_v3_simple_3 (Post-Revising)
def solve_quadratic(a : List[float], r : float) -> Tuple[float, float]:
    """[...]"""
    # Importing the custom module
    from custom import quadratic_roots
    # Extracting the coefficients from the list 'a'
    coefficient_a = a[0]
    coefficient_b = a[1]
    constant = a[2]
    # Using the quadratic_roots function to find the solutions
    x1, x2 = quadratic_roots(coefficient_a, coefficient_b, constant - r)
    # Returning the solutions as a Tuple
    return (x1, x2)

# Task: BFCL_v3_simple_3 (RoCode)
def solve_quadratic(a : List[float], r : float) -> Tuple[float, float]:
    """[...]"""
    import custom
    roots = custom.quadratic_roots(a[0], a[1], a[2] - r)
    return roots

# Task: BFCL_v3_simple_3 (Solution)
def solve_quadratic(a : List[float], r : float) -> Tuple[float, float]:
    """[...]"""
    from custom import quadratic_roots
    return quadratic_roots(a=a[0], b=a[1], c=a[2] - r)
```

Die generierten Methoden der analysierten Ansätze mit dem Sprachmodell Qwen2.5-Coder-3B-Instruct für eine an *BFCL_v3_simple_3* (vgl. [36]) angelehnte Aufgabe.

B Generierte Antworten für BFCL_v3_simple_115

```python
# Task: BFCL_v3_simple_115 (Temperature-Sampling)
def calculate_coinflip(tosses: int, max_heads : int) -> float:
    """
    Calculates the propability of tossing at max (inclusive) [max_heads] heads on [tosses
        ] tosses using a fair (50/50) coin.
    This function utilizes 'calculate_binomial_propability' from the 'custom' module.

    >>> calculate_coinflip(10, 3)
    0.17188
    >>> calculate_coinflip(7, 7)
    1.0
    >>> calculate_coinflip(15, 13)
    0.99951
    """
    from custom import calculate_binomial_propability
    return calculate_binomial_propability(tosses, max_heads, 0.5)

# Task: BFCL_v3_simple_115 (AdapT)
def calculate_coinflip(tosses: int, max_heads : int) -> float:
    """[...]"""
    from custom import calculate_binomial_propability
    return calculate_binomial_propability(tosses, max_heads, 0.5)

# Task: BFCL_v3_simple_115 (Context-Adaptive Sampler)
def calculate_coinflip(tosses: int, max_heads : int) -> float:
    """[...]"""
    from custom import calculate_binomial_propability
    return calculate_binomial_propability(tosses, 2, max_heads)

# Task: BFCL_v3_simple_115 (Sampling & Filtering)
def calculate_coinflip(tosses: int, max_heads : int) -> float:
    """[...]"""
    # Importing the custom module and its function
    from custom import calculate_binomial_propability
    # Calculating the binomial probability for max_heads heads and tosses total
    prob = calculate_binomial_propability(tosses, max_heads, 0.5)
    # Returning the calculated probability
    return prob

# Task: BFCL_v3_simple_115 (Post-Revising)
def calculate_coinflip(tosses: int, max_heads : int) -> float:
    """[...]"""
    from custom import calculate_binomial_propability
    return calculate_binomial_propability(tosses, max_heads, 0.5)

# Task: BFCL_v3_simple_115 (RoCode)
def calculate_coinflip(tosses: int, max_heads : int) -> float:
```

```
    """[...]"""
    # Import calculate_binomial_propability from custom module
    from custom import calculate_binomial_propability

    # Calculate the total number of possible outcomes (2^tosses)
    total_outcomes = 2 ** tosses
    total_propabilities = 0
    for heads in range(max_heads + 1):
        # Calculate the binomial probability of getting 'heads' heads in 'tosses' tosses
        probability = calculate_binomial_propability(heads, tosses, 0.5)
        total_propabilities += probability
    # Return the total probability divided by the total number of outcomes
    return total_propabilities / total_outcomes

# Task: BFCL_v3_simple_115 (Solution)
def calculate_coinflip(tosses: int, max_heads : int) -> float:
    """[...]"""
    from custom import calculate_binomial_propability
    return sum(calculate_binomial_propability(k=k, n=tosses, p=0.5) for k in range(0,
        max_heads + 1))
```

Die generierten Methoden der analysierten Ansätze mit dem Sprachmodell Qwen2.5-Coder-3B-Instruct für eine an $BFCL\_v3\_simple\_115$ (vgl. [36]) angelehnte Aufgabe.

C Generierte Antworten für BFCL_v3_simple_144

```python
# Task: BFCL_v3_simple_144 (Temperature-Sampling)
def get_stock_advance(company : str, from_year : int, to_year : int) -> float:
    """
    Calculates the percentage increase / decrease of stock value from company [company]
        over the specified time period.
    Stock data is fetched using 'get_stock_value' from the 'custom' module.

    >>> get_stock_advance("Apple", 2005, 2014)
    0.48539
    >>> get_stock_advance("Meta", 2006, 2012)
    0.10953
    >>> get_stock_advance("Davinion", 2014, 2016)
    -0.24286
    """
    from custom import get_stock_value
    from_year_value = get_stock_value(company, from_year)
    to_year_value = get_stock_value(company, to_year)
    return (to_year_value - from_year_value) / from_year_value * 100

# Task: BFCL_v3_simple_144 (AdapT)
def get_stock_advance(company : str, from_year : int, to_year : int) -> float:
    """[...]"""
    stock_values = []
    for year in range(from_year, to_year + 1):
        stock_values.append(get_stock_value(company, year))
    initial_value = stock_values[0]
    final_value = stock_values[-1]
    if initial_value == 0:
         return 0
    else:
         return ((final_value - initial_value) / initial_value) * 100

# Task: BFCL_v3_simple_144 (Context-Adaptive Sampler)
def get_stock_advance(company : str, from_year : int, to_year : int) -> float:
    """[...]"""
    # Fetch the stock values for the given company and time period
    stock_values = get_stock_value(company, from_year, to_year)

    # Calculate the average stock value for the given time period
    average_stock_value = sum(stock_values) / len(stock_values)

    # Fetch the stock value for the first year in the specified time period
    initial_stock_value = stock_values[0]

    # Calculate the percentage increase / decrease
    percentage_change = ((average_stock_value - initial_stock_value) /
        initial_stock_value) * 100
```

```
    # Return the percentage increase / decrease
    return percentage_change

# Task: BFCL_v3_simple_144 (Sampling & Filtering)
def get_stock_advance(company : str, from_year : int, to_year : int) -> float:
    """[...]"""
    from custom import get_stock_value
    current_year = to_year
    previous_year = from_year
    current_value = get_stock_value(company, current_year)
    previous_value = get_stock_value(company, previous_year)
    advance = (current_value - previous_value) / previous_value
    return advance

# Task: BFCL_v3_simple_144 (Post-Revising)
def get_stock_advance(company : str, from_year : int, to_year : int) -> float:
    """[...]"""
    from custom import get_stock_value

    company_stock_values = []
    for year in range(from_year, to_year + 1):
        stock_value = get_stock_value(company, year)
        company_stock_values.append(stock_value)

    if len(company_stock_values) < 2:
        raise ValueError("Company stock data for the specified years is insufficient.")

    initial_value = company_stock_values[0]
    final_value = company_stock_values[-1]
    percentage_advance = ((final_value - initial_value) / initial_value) * 100

    return percentage_advance

# Task: BFCL_v3_simple_144 (RoCode)
def get_stock_advance(company : str, from_year : int, to_year : int) -> float:
    """[...]"""
    from custom import get_stock_value
    start_value = get_stock_value(company, from_year)
    end_value = get_stock_value(company, to_year)
    if end_value == 0:
        return 0
    return ((end_value - start_value) / start_value) * 100

# Task: BFCL_v3_simple_144 (Solution)
def get_stock_advance(company : str, from_year : int, to_year : int) -> float:
    """[...]"""
    from custom import get_stock_value
    return get_stock_value(company=company, year=to_year) / get_stock_value(company=
```

```
        company, year=from_year) - 1
```

Die generierten Methoden der analysierten Ansätze mit dem Sprachmodell Qwen2.5-Coder-3B-Instruct für eine an $BFCL\_v3\_simple\_144$ (vgl. [36]) angelehnte Aufgabe.

D Generierte Antworten für BFCL_v3_parallel_65

```python
# Task: BFCL_v3_parallel_65 (Temperature-Sampling)
def get_circumferences(radii : List[float]) -> List[float]:
    """
    Calculates the circumferences of the circiles with the specified radii.
    This function utilizes 'calculate_circumference' from the 'custom' module.

    >>> get_circumferences([2,6,7,1])
    [12.56637, 37.69911, 43.98229, 6.28318]
    >>> get_circumferences([5, 9, 0])
    [31.41592, 56.54866, 0.0]
    >>> get_circumferences([15])
    [94.24777]
    """
    return [calculate_circumference(radius) for radius in radii]

# This is a helper function for the get_circumferences function.
def calculate_circumference(radius : float) -> float:
    """
    Calculates the circumference of a circle with the given radius.
    This function utilizes the formula 2 * pi * radius.

    >>> calculate_circumference(2)
    12.56637
    >>> calculate_circumference(9)
    56.54866
    >>> calculate_circumference(0)
    0.0
    """
    return 2 * 3.141592 * radius

# Task: BFCL_v3_parallel_65 (AdapT)
def get_circumferences(radii : List[float]) -> List[float]:
    """[...]"""
    import custom
    return [custom.calculate_circumference(r) for r in radii]

# Task: BFCL_v3_parallel_65 (Context-Adaptive Sampler)
def get_circumferences(radii : List[float]) -> List[float]:
    """[...]"""
    return [calculate_circumference(radius) for radius in radii]

# Task: BFCL_v3_parallel_65 (Sampling & Filtering)
def get_circumferences(radii : List[float]) -> List[float]:
    """[...]"""
    return [calculate_circumference(radius) for radius in radii]

# This is a placeholder for the 'calculate_circumference' function.
```

```python
# It is assumed to be defined in a module named 'custom'.
# For the purpose of this example, I will define a simple function.
def calculate_circumference(radius: float) -> float:
    return 2 * 3.14159 * radius

# Task: BFCL_v3_parallel_65 (Post-Revising)
def get_circumferences(radii : List[float]) -> List[float]:
    """[...]"""
    from custom import calculate_circumference  # Import the calculate_circumference
        function from the custom module
    circumferences = []
    for radius in radii:
        # Ensure the radius is not negative, as the circumference cannot be calculated
            for negative radii
        if radius < 0:
            raise ValueError("Negative radii are not allowed.")
        # Call the calculate_circumference function from the custom module
        circumference = calculate_circumference(radius)
        circumferences.append(circumference)
    return circumferences

# Task: BFCL_v3_parallel_65 (RoCode)
def get_circumferences(radii : List[float]) -> List[float]:
    """[...]"""
    def calculate_circumference(radius : float) -> float:
        return 2 * 3.14159 * radius
    return [calculate_circumference(radius) for radius in radii]

# Task: BFCL_v3_parallel_65 (Solution)
def get_circumferences(radii : List[float]) -> List[float]:
    """[...]"""
    from custom import calculate_circumference
    return [calculate_circumference(diameter=r*2) for r in radii]
```

Die generierten Methoden der analysierten Ansätze mit dem Sprachmodell Qwen2.5-Coder-3B-Instruct für eine an *BFCL_v3_parallel_65* (vgl. [36]) angelehnte Aufgabe.

E Generierte Antworten für BFCL_v2_parallel_67

```python
# Task: BFCL_v2_parallel_67 (Temperature-Sampling)
def get_largest_circle(radii : List[float]) -> int:
    """
    Identifies the larges circle by area and returns it's index.
    For area calculation, 'calculate_circle_area' is used from the 'custom' module.

    >>> get_largest_circle([3, 5, 7, 0, 2])
    2
    >>> get_largest_circle([4, 8, 1])
    1
    >>> get_largest_circle([7, 9, 25, 15])
    2
    """
    largest_index = 0
    largest_area = calculate_circle_area(radii[0])
    for i in range(1, len(radii)):
        area = calculate_circle_area(radii[i])
        if area > largest_area:
            largest_index = i
            largest_area = area
    return largest_index

# Task: BFCL_v2_parallel_67 (AdapT)
def get_largest_circle(radii : List[float]) -> int:
    """[...]"""
    # Import the calculate_circle_area function from the custom module
    from custom import calculate_circle_area

    # Initialize variables to track the largest area and its index
    largest_area = 0
    largest_index = -1

    # Iterate over the radii to find the largest circle
    for index, radius in enumerate(radii):
        # Calculate the area of the current circle
        current_area = calculate_circle_area(radius)

        # Update the largest area and index if the current area is larger
        if current_area > largest_area:
            largest_area = current_area
            largest_index = index

    # Return the index of the largest circle
    return largest_index

# Task: BFCL_v2_parallel_67 (Context-Adaptive Sampler)
def get_largest_circle(radii : List[float]) -> int:
```

```
    """[...]"""
    def calculate_circle_area(radius: float) -> float:
        import math
        return math.pi * radius * radius

    # Find the index of the largest circle based on area
    largest_index = 0
    largest_area = 0

    for index, radius in enumerate(radii):
        area = calculate_circle_area(radius)
        if area > largest_area:
            largest_area = area
            largest_index = index

    return largest_index

# Task: BFCL_v2_parallel_67 (Sampling & Filtering)
def get_largest_circle(radii : List[float]) -> int:
    """[...]"""
    # Importing the necessary function from a custom module
    from custom import calculate_circle_area

    # Calculate the area of each circle
    areas = [calculate_circle_area(radius) for radius in radii]

    # Find the index of the largest area
    largest_index = areas.index(max(areas))

    # Return the index of the largest circle
    return largest_index

# Task: BFCL_v2_parallel_67 (Post-Revising)
def get_largest_circle(radii : List[float]) -> int:
    """[...]"""
    # Importing the calculate_circle_area function from the custom module
    from custom import calculate_circle_area

    # Initializing variables to track the largest area and its index
    largest_area = float('-inf')
    largest_index = -1

    # Iterating through each radius to find the largest circle
    for i, radius in enumerate(radii):
        # Calculating the area of the current circle
        area = calculate_circle_area(radius)

        # Updating the largest area and index if the current circle's area is larger
        if area > largest_area:
```

```
            largest_area = area
            largest_index = i

    # Returning the index of the largest circle
    return largest_index

# Task: BFCL_v2_parallel_67 (RoCode)
def get_largest_circle(radii : List[float]) -> int:
    """[...]"""
    # Import the calculate_circle_area function from the custom module
    from custom import calculate_circle_area
    # Initialize the variable to track the index of the largest circle
    largest_circle_index = 0
    # Initialize the variable to track the maximum area found
    max_area = 0
    # Iterate over each radius in the list
    for i, radius in enumerate(radii):
        # Calculate the area of the circle with the current radius
        area = calculate_circle_area(radius)
        # Check if the current area is greater than the maximum area found
        if area > max_area:
            # Update the maximum area and the index of the largest circle
            max_area = area
            largest_circle_index = i
    # Return the index of the largest circle
    return largest_circle_index

# Task: BFCL_v2_parallel_67 (Solution)
def get_largest_circle(radii : List[float]) -> int:
    """[...]"""
    return radii.index(max(radii))
```

Die generierten Methoden der analysierten Ansätze mit dem Sprachmodell Qwen2.5-Coder-3B-Instruct für eine an *BFCL_v2_parallel_67* (vgl. [36]) angelehnte Aufgabe.

F Generierte Antworten für den Praxistest

```python
from arango.client import ArangoClient

# Task: Praxistest (Temperature-Sampling)
def get_arango_client(hostname : str, port : Optional[int] = None) -> ArangoClient:
    """ Generates a new arango client, that connects to the specified address.
    When no port is provided, the default port (8529) will be used.

    The generated ArangoClient instanze will be disconnected and thus requires a 'connect
        ' call before proper use.
    """
    # Your implementation here
    client = ArangoClient(
        host=hostname,
        port=port or 8529  # Use default port 8529 if port is not provided
    )
    return client

# Task: Praxistest (AdapT)
def get_arango_client(hostname : str, port : Optional[int] = None) -> ArangoClient:
    """ [...]"""
    if port is None:
         port = 8529

    client = ArangoClient(
        host=hostname,
        port=port
    )

# Task: Praxistest (Context-Adaptive Sampler)
def get_arango_client(hostname : str, port : Optional[int] = None) -> ArangoClient:
    """[...]"""
    if port is None:
        port = 8529
    return ArangoClient(hosts=[f"{hostname}:{port}"])

# Task: Praxistest (Sampling & Filtering)
def get_arango_client(hostname : str, port : Optional[int] = None) -> ArangoClient:
    """[...]"""
    if port is None:
        port = 8529
    return ArangoClient(hosts=f"{hostname}:{port}")

# Task: Praxistest (Post-Revising)
def get_arango_client(hostname : str, port : Optional[int] = None) -> ArangoClient:
    """[...]"""
    arango_client = ArangoClient(
        hosts=[hostname],
```

```
        port=port or 8529
    )

# Task: Praxistest (RoCode)
def get_arango_client(hostname : str, port : Optional[int] = None) -> ArangoClient:
    """[...]"""
    if port is None:
        port = 8529
    return ArangoClient(hosts=[f"{hostname}:{port}"])

# Task: Praxistest (Solution)
def get_arango_client(hostname : str, port : Optional[int] = None) -> ArangoClient:
    """[...]"""
    return ArangoClient(hosts=f'{hostname}:{port if port is not None else 8529}')
```

Die generierten Methoden der analysierten Ansätze mit dem Sprachmodell Qwen2.5-Coder-3B-Instruct für einen Praxistest, der maßgeblich durch bestehenden Code am DLR und gemachten Erfahrungen bei der autonomen Codegenerierung beeinflusst wurde.