

Generative KI Large Language Models in Softwareentwicklung und Softwaresicherheit

Florian Reichle, B.Sc.

Abstract

The work presents opportunities and challenges of using generative AI in the software development process and provides users with guidance on how to avoid problems inherent to generative AI.

Die vorliegende Arbeit präsentiert die Möglichkeiten und Herausforderungen der Nutzung von generativer KI im Softwareentwicklungsprozess. Sie gibt Nutzenden Empfehlungen zur Nutzung von generativer KI und bietet Orientierungshilfe um Probleme zu vermeiden.

Index Terms

GenAI, Large Language Model, Software Development Life Cycle, Small Language Model, Security

Juli 14, 2025

I. GRUNDLAGEN: GENERATIVE KI UND LARGE LANGUAGE MODELS (LLM)

Generative KI, ein Teilbereich der unter dem Oberbegriff Künstliche Intelligenz zusammengefassten “intelligenten” Computersysteme, umfasst Prozesse und Algorithmen, welche in der Lage sind, aus Mustern und Strukturen bestehender Daten neue Inhalte zu generieren. Dafür werden die Algorithmen anhand einer Vielzahl bestehender Datensätze trainiert. Die generative KI “lernt” so quasi selbständig aus den, den vorhandenen Daten hinterliegenden Strukturen und Mustern. Die generative KI wendet diese Strukturen und Muster später dann an, um neue, nicht in den Ursprungsdaten vorhandene Ergebnisse zu produzieren. Beispielsweise könnte ein Algorithmus einer generativen KI, der mit Vogelbildern trainiert wurde, ein völlig neues Bild eines Vogels generieren. Dabei würde dieser “neue” Vogel die Charakteristiken der Trainingsvogelbilder zwar widerspiegeln, jedoch keinem Vogel genau entsprechen.

A. *Large Language Models (LLM)*

Large Language Models sind spezifische Typen generativer KI, die mithilfe einer Vielzahl von Textdatensätzen trainiert wurden. Sie sind in der Lage, die Muster und den Aufbau von Sprache mathematisch auszudrücken und können Texte sowohl “verstehen” als auch neue Texte generieren. Hierbei arbeiten sie mit Wortsequenzen. Dabei wird das jeweils nachfolgende Wort eines Satzes anhand der Reihenfolge und der Häufigkeit der Verwendung in Sätzen oder größeren Textpassagen in den Trainingsdaten ermittelt. So konnten LLMs - wie zum Beispiel das durch Presseveröffentlichungen auch in der Bevölkerung bekannte Chat-GPT - innerhalb kürzester Zeit durch die Fähigkeit menschenähnliche Texte zu generieren, Schlagzeilen machen und erfreuen sich allgemeiner Beliebtheit.

B. *Small Language Models (SLM)*

Small Language Models sind, wie ihr Name bereits vermuten lässt, quasi eine kleinere Version der LLMs. Sie nutzen ein deutlich kleineres Datenvolumen (oft um den Faktor 100 geringer) als LLMs, sind jedoch trotzdem in der Lage, Sprache zu “verstehen” und Texte zu generieren. Dementsprechend sind sie deutlich ressourcenschonender und dadurch auch für ressourcenarme Systeme geeignet. Trotz ihrer infolge der geringeren Größe geringeren allgemeinen Fähigkeiten bieten sich SLMs insbesondere an, um

spezifische Probleme zu bearbeiten. Sie sollten jedoch jeweils genau für diese Problemstellungen trainiert und verwendet werden. “By using smaller language models, the same level of success can be achieved with fewer parameters and less energy consuming hardware.” [1]

C. Training

Der wichtigste Schritt bei der Erstellung und Verwendung von generativer KI ist das jeweilige Training. Als Training bezeichnet man dabei den Prozess, einen Algorithmus ohne eigene, manuelle Algorithmusstellung mithilfe vorhandener Datensätze zu erzeugen. Hier gibt es verschiedene Methoden:

1) *Supervised Learning*: Beim Supervised Learning sind die jeweiligen Trainingsdaten beschriftet und bilden jeweils vergangene Beobachtungen ab. Hier lernt der Algorithmus durch einen “Lehrer”, der bestimmte Dinge vorgibt und den Prozess aktiv beeinflusst. Diese Technik wird häufig verwendet, um unbekannte Werte einordnen zu können. Beispielsweise wird dies in der Erkennung von Spam verwendet. Der Algorithmus untersucht dafür bereits als Spam bzw. nicht-Spam markierte E-Mails und verwendet diese Informationen sowie die Verwendung bestimmter Wörter, Absender, den Betreff oder die Länge der E-Mail, um neue E-Mails als Spam zu klassifizieren.

2) *Unsupervised Learning*: Beim Unsupervised Learning sind dagegen die Trainingsdaten nicht beschriftet oder die Probleme sind zu komplex, um von einem menschlichen “Lehrer” analysiert und verstanden zu werden. In diesen Fällen lernt der Algorithmus eigenständig, nur durch Beobachtung und erkennt im besten Fall Muster. Ein gutes Beispiel hierfür sind die Algorithmen, die für das Musikstreaming verwendet werden bzw. zur Erzeugung der Empfehlung von Produkten durch Recommendation Engines. Im ersten Beispiel beobachtet der jeweilige Algorithmus, welche Songs oft nacheinander oder im Zusammenhang gehört werden und gruppiert diese, um sie später weiterzuempfehlen, wohingegen im zweiten Beispiel oft zusammen gekaufte Produkte empfohlen werden.

3) *Reinforcement Learning*: Diese dritte Kategorie des Lernens beschreibt das zielbasierte Training. Hierbei ist der Weg nicht definiert. Entscheidend ist nur, wie gut das Ziel erreicht wurde. Hier lernt der jeweilige Algorithmus durch Ausprobieren und ihm nach einer Entscheidung zurückgespiegeltes Feedback, das er jeweils verwendet, um gegebenenfalls nachfolgende, künftige Entscheidungen zu korrigieren. Ein Beispiel hierzu in Appendix A. Diese Technik kann unter anderem verwendet werden, um einem Computer ein Spiel wie Schach beizubringen. [2]

4) *Self-supervised Learning*: Supervised Learning-Methoden lernen häufig besser und schneller. Sie erfordern jedoch wesentlich mehr beschriftete Daten und Datensätze, die jeweils in zeitaufwändiger Handarbeit erstellt werden müssen. [3] Beim Self-supervised Learning wird daher versucht, das Beste aus Supervised und Unsupervised Learning Methoden zu kombinieren. Hierbei erstellt sich der Algorithmus selbst Trainingsdaten aus bereits existierenden Echtdaten. Beispielsweise für den Text “Heute ist das Wetter schön” würde der Algorithmus jeweils ein Wort maskieren und jeweils einen Datenpunkt für dieses erstellen. Siehe hier eine tabellarische Darstellung:

Satz	Korrekte Lösung
(Maskiert) ist das Wetter schön	Heute
Heute (Maskiert) das Wetter schön	ist
Heute ist (Maskiert) Wetter schön	das
Heute ist das (Maskiert) schön	Wetter
Heute ist das Wetter (Maskiert)	schön

TABLE I: Word Mask

Anschließend versucht der Algorithmus jeweils den Satz zu vervollständigen und lernt mithilfe der bekannten korrekten Lösung die Genauigkeit der Vorhersage zu bestimmen.

II. ARCHITEKTUR VON LLMs

Bereits in den 1950er Jahren wurde anhand von Studien die semantische Struktur von Sprache untersucht und diese Forschung seitdem stetig verbessert. So erlaubt Word2Vec [4], eine von Google 2013 veröffentlichte Bibliothek, Sprache mithilfe von Vektoren numerisch darzustellen und so in Berechnungen zu verwenden.

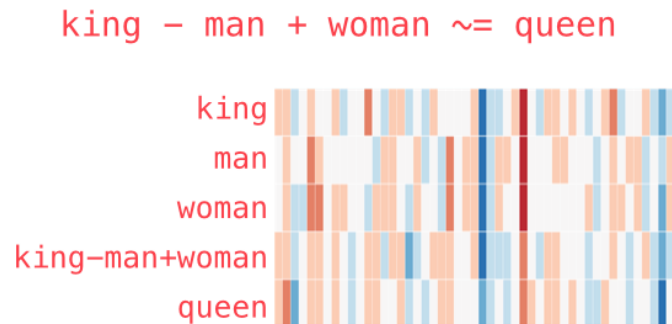


Fig. 1: King Analogy W2Vec [5]

Forschungen, basierend auf diesen Technologien, mündeten schließlich in der Transformer Technologie, welche 2017 von Google in “Attention Is All You Need” [6] veröffentlicht wurde und es erlaubt, auch komplexe Beziehungen wie Kontextabhängigkeit von Wörtern zu verstehen.

A. Token

Bevor maschinelles Lernen dazu in der Lage ist, Text zu verarbeiten, muss der jeweilige Text zuerst in eine numerische Form gebracht werden. LLMs, wie Chat-GPT, verwenden hierfür häufig zusammen auftretende Zeichenketten, sogenannte Tokens, zur internen Textverarbeitung. Nachfolgend dargestellt ein Text und wie dieser von GTP-4o [7] in einzelne Tokens aufgespalten wird:

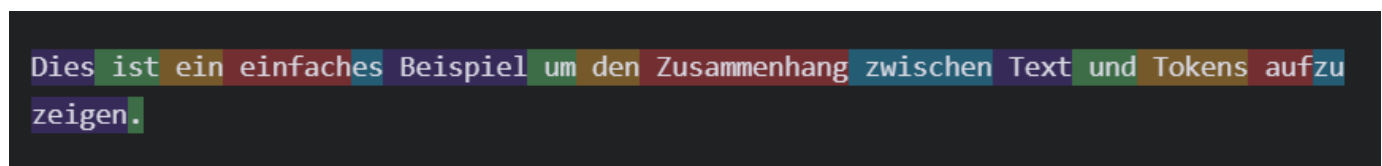


Fig. 2: Tokens [7]

Häufig auftretende Zeichenketten wie “es” und “zu” sind dabei jeweils einzelne Token, wohingegen längere Wörter wie “aufzuzeigen”, sich aus mehreren häufiger auftretenden Tokens zusammensetzen lassen und in dem Beispiel in die verschiedenen Tokens (“auf”, “zu”, “zeigen”) aufgespalten werden. Diese Tokens werden als numerische Werte von der LLM verarbeitet. Hier der gleiche Text nur als Array der numerischen Tokenwerte:

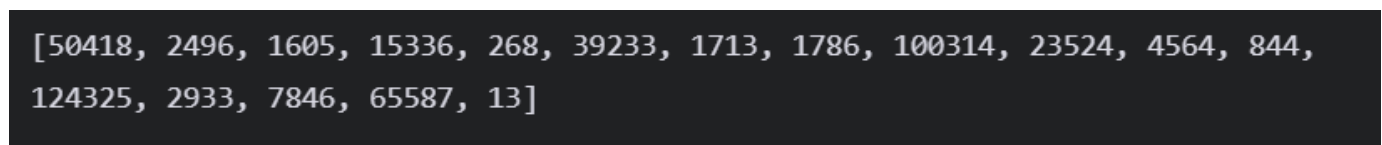


Fig. 3: Tokens numerisch [7]

Hier sei darauf aufmerksam gemacht, dass häufig auftretenden Tokens wie “.” (13), “es” (268) sowie “und” (844) niedrige Werte zugeordnet werden, wohingegen selten auftretende Wörter wie “Zusammen-

hang” (100314) und “Tokens” (124325) größere numerische Werte erhalten, was die spätere Effizienz der Berechnung steigert.

B. Selbstaufmerksamkeit

Der Begriff Selbstaufmerksamkeit beschreibt im KI-Kontext Algorithmen, deren Aufgabe es ist, einem KI-Modell mitzuteilen, welchem Teil der eingegebenen Daten die größte Aufmerksamkeit geschenkt werden soll. Am Beispiel des Satzes “Das Wetter ist heute schön” sei dies dargestellt:

- Der Artikel “Das” impliziert, dass das Datum Wetter ein Nomen ist.
- Dieses Nomen beschreibt den jeweiligen Zustand der Atmosphäre zu einem bestimmten Zeitpunkt [8] und ist nicht eine Umschreibung für laut und heftig schimpfen. [9]
- “Wetter” ist dabei innerhalb des Satzes das einzige Nomen und legt das Thema des Satzes fest.
- “heute” beschreibt, dass es sich um den Zustand der Atmosphäre an einem bestimmten Zeitpunkt handelt.
- “schön” beschreibt die Eigenschaft des Zustands und bezieht sich im Zusammenhang von Wetter meist auf Sonnenschein mit wenig Wolken

Durch die vorgegebene Selbstaufmerksamkeit “weiß” das KI-Modell, dass es sich besonders auf die Wörter “Wetter”, “heute” und “schön” konzentrieren muss, um den Satz inhaltlich verstehen zu können. Der Artikel “das” ist für den Sinn nur wenig relevant und “ist” ist nur ein Hilfswort, das wenig semantische Informationen liefert.

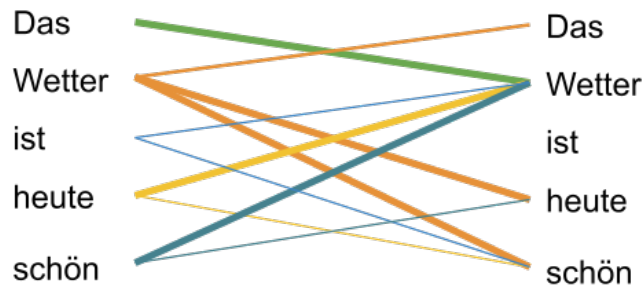


Fig. 4: Beispiel Selbstaufmerksamkeit

C. Temperature

LLMs berechnen Token anhand der bereits analysierten Worte/Datensätze. Hierbei folgen sie den Wahrscheinlichkeiten der Datensätze, mit denen vorher trainiert worden ist. Bei unserem Beispiel könnte so der Text “Heute ist das Wetter...” von der KI wie folgt vervollständigt werden:

Token	Wahrscheinlichkeit
schön	65%
sonnig	20%
regnerisch	10%
grausam	3%
elektronisch	1%
blau	1%

TABLE II: Token-Wahrscheinlichkeit

Die “temperature” einer LLM kontrolliert jeweils den Zufall der Tokenauswahl. Hierbei sind niedrige “temperature” besser für deterministische Ergebnisse, wohingegen höhere “temperature” besser für diversifizierte und unerwartete Ergebnisse sind. Eine “temperature” von 0.0 würde hier immer das Wort mit der höchsten Wahrscheinlichkeit wählen. Eine mittlere “temperature” - z.B. von 0.5 - würde kreativere, aber immer noch kontrollierte Ergebnisse liefern, könnte den Satz also sowohl mit “schön”, “sonnig” oder “regnerisch” vervollständigen. Da aber die Wahrscheinlichkeiten der Wörter immer noch eine Rolle spielen und diese das Ergebnis anhand der Wahrscheinlichkeiten “würfelt”, würde also deutlich häufiger passieren, dass das Wetter als schön beschrieben wird, als als regnerisch. Bei einer hohen “temperature” - z.B. 1.0 - könnten jedoch auch sehr unwahrscheinliche Wörter auftreten und bei der Frage nach dem heutigen Wetter auf einmal mit der Beschreibung “blau” geantwortet werden.

D. Halluzination

Doch auch bei mittleren oder niedrigen “temperature” besteht die Gefahr, dass die KI unsinnige Aussagen erzeugt oder falsche Rückschlüsse zieht, dies jedoch als faktische Aussagen darstellt. Dies ist auch als “KI-Halluzination” bekannt. Bei wenig bekannten Daten sowie bei Daten, die sich häufig ändern oder schlecht dokumentiert sind, steigt die Gefahr dieses Problems exponentiell. Beispielhaft hier die Anfrage an die KI: “Wie kann ich mein Netflix-Abo im Jahr 2025 kündigen?” Hierauf würde die KI antworten: “Gehe in dein Profil, wähle Mein Konto und klicke auf Abo kündigen ganz unten auf der Seite.” Diese Aussage klingt plausibel und ist auch faktisch richtig. Das Problem ist jedoch, dass Netflix jederzeit sein Interface umstellen oder den Abo-kündigen-Prozess unter einem anderen Unterpunkt ansiedeln könnte, ohne dass dies bei der Antwort der KI berücksichtigt würde. Bis sich diese neuen Informationen als die “wahrscheinlichste Antwort” etablieren würden, wären diese eventuell bereits wieder veraltet. [10]

E. Prompt

LLMs sind dafür entwickelt, auf Eingabe von Nutzenden sogenannte Prompts zu reagieren und darauf passende Antworten zu generieren. Wie bereits erörtert, sind deren Antworten nicht perfekt. Je spezifischer der Prompt jedoch formuliert ist und je mehr Details dem Prompt hinzugefügt werden, desto besser kann die LLM die Antwort vorhersagen. Nutzende, die sich diesen Limitierungen und der unterliegenden LLM-Funktionalität bewusst sind, können dann spezielle Techniken anwenden, um dies zu ihrem Vorteil zu nutzen. Dies wird allgemein als “Prompt Engineering” bezeichnet.

F. Prompt Engineering

Google hat zum Prompt Engineering im Februar 2025 ein Whitepaper über die gängigsten Taktiken und Regeln veröffentlicht, in welchem Lee Boonstra den iterativen Prozess spezifisch für Google Gemini beschreibt. [11] Generell sind diese Techniken modellunspezifisch und können so allgemein bei der Verwendung von LLMs angewendet werden. Im Nachfolgenden daher ein kurzer Auszug der wichtigsten von Boonstra genannten Techniken:

1) *General Prompting/Zero Shot*: Generelles Prompten - oder “Zero Shot Prompts” - ist generell die einfachste Art der Prompt-Anwendung. Der Begriff bezieht sich auf die Abwesenheit von Beispielen (shots). Hierbei handelt es sich um das Stellen einer Frage oder das Beschreiben einer Aufgabe - ähnlich einer Google-Anfrage. Ein Beispiel hierzu in Appendix B.

2) *One-Shot & Few-Shot*: Bei komplexeren Fragestellungen bietet es sich jedoch oft an, Probleme mit Beispielen zu erläutern. Da KI-Modelle mithilfe öffentlich zugänglicher Daten trainiert werden, bietet sich dies auch beim Erstellen von Prompts an, da diese der KI dabei behilflich sein können, das Problem genauer zu verstehen bzw. auch als Grundlage für das Vorhersagen der Antwort dienen kann. Dabei sind sie dann besonders hilfreich, wenn man eine spezifisch strukturierte Ausgabe erhalten möchte. Je mehr Beispiele der Prompt enthält, desto detaillierter wird dann das Ergebnis. Zu erkennen in Appendix C.

3) *System-, kontextuelles und Rollen-Prompten*: Alternativ bzw. zusätzlich zu dem Verwenden von Beispielen gibt es weitere Techniken, die verwendet werden können, um einer LLM Vorgaben zur verbesserten Textgenerierung zu geben. Diese können grob in 3 Subtechniken unterteilt werden:

a) *System Prompting*: System Prompting nutzt zur Verbesserung den Kontext und Zweck des jeweiligen LLM-Modells. Hierbei wird der grobe Rahmen der jeweiligen Aufgabe beschrieben und als Vorgabe gesetzt. Im nachfolgenden Beispiel wird der Unterschied zum Weglassen des Systemprompts deutlich. Da LLMs oft Erklärungen hinzufügen, entfallen durch das Beschreiben des Aufgabenrahmens jedwede Erklärung sowie Codekommentare. Siehe Appendix D.

b) *Contextual Prompting*: Beim kontextualen Prompten werden dem Prompt weitere Hintergrundinformationen bzw. Details hinzugefügt, die für das Erreichen der gestellten Aufgabe relevant sind. Hierdurch kann die LLM ihre Antwort präzisieren, indem sie mit dieser nun genaueren Beschreibung des Ziels ihre Berechnungen durchführt. Im folgenden Beispiel lässt die LLM beispielsweise die Main Methode unbeachtet, da diese für das Ziel "Ausgeben von "Hello World"" in Python nicht notwendig und zum Erreichen des Ziels des Programmcodes unerheblich ist. Zu erkennen in Appendix E.

c) *Role Prompting*: Bei Nutzung des Role Prompting weist der Prompt-Erstellende der LLM eine bestimmte Rolle zu. Hierdurch wird sichergestellt, dass die LLM die Fragestellung nur aus der Sicht dieser Person betrachtet wie in Appendix F. Diese Technik kann von Nutzenden jedoch auch verwendet werden, um die vom LLM-Hersteller eingebauten Sicherheitsmechanismen zu umgehen. So können die anfänglich unter dem Namen DAN ("Do Anything Now") auf Reddit verbreiteten Techniken [12] beispielsweise auch verwendet werden, um eine LLM dazu zu nutzen, ein eigentlich nicht zulässiges Ergebnis - wie eine Anleitung zum Bau einer Bombe - zu erzeugen. [13]

d) *Step Back Prompten*: Beim Step Back Prompting wird jeweils vor die eigentliche Fragestellung eine allgemeinere Frage vorangestellt. Die Antwort dieser vorangestellten Frage wird dabei verwendet, um die nachfolgende, eigentliche Fragestellung in einen verbesserten Prompt umzuwandeln, der dann die eigentlich geforderte Antwort in einer verbesserten Version erzeugt. Durch das Voranstellen einer weiteren Frage erhält die LLM die Möglichkeit, weitere Hintergrundinformationen hinzuzufügen, bevor sie dann beginnt, das eigentliche Problem zu lösen. Ein Beispiel in Appendix G.

e) *Chain of Thought*: Die Methode "Chain of Thought", auch oft als "Reasoning" bezeichnet, beschreibt eine Methode, bei der die LLM dazu angehalten wird, ihren "Denkprozess" logisch und aufeinander aufbauend zu gestalten. Zur Nutzung dieser Methode bieten derzeit populäre LLMs wie Chat-GPT eine eigene "Reasoning"-Funktion: Siehe Appendix H. Ein im Juni 2025 von Apple zu dieser Methode veröffentlichtes Papier [14] bezweifelt jedoch den Vorteil der Anwendung von "Reasoning". So seien bei niedrig komplexen Problemstellungen oft die Ergebnisse des LLM-Modells ohne die Anwendung der "Reasoning"-Funktion besser. Auch bei komplexen Problemen bietet die Anwendung von "Reasoning" keinen Vorteil: "[...] standard LLMs outperform LRMs at low complexity, LRMs excel at moderate complexity, and both collapse at high complexity."

III. SICHERHEITSRELEVANTE EIGENSCHAFTEN

Um LLM-Algorithmen das Erkennen von Mustern und das Treffen von Vorhersagen “beizubringen”, muss durch die Eingabe einer möglichst großen Datenmenge dies zunächst ermöglicht werden. Je vielfältiger und repräsentativer die jeweiligen Daten dabei sind, desto besser kann das Modell verallgemeinern und auf dieser Basis neue Schlüsse ziehen. Verallgemeinernd gilt daher die Feststellung: Eine KI ist nur so gut wie die Daten, mit denen der zugrundeliegende Algorithmus trainiert wurde. Vergleiche [15].

Dies ist besonders bei der Generierung von Codes relevant. Eine 2023 von gitguardian veröffentlichte Studie kommt beispielsweise zu der Erkenntnis, dass 2022, im Jahr der Untersuchung, 67 % mehr im Sourcecode befindende API-Schlüssel und andere schützenswerte Daten auf GitHub gelandet sind, als in den Jahren zuvor. Dies, obwohl im Jahr 2022 20 % mehr Quellcodes analysiert wurden. [16] Die gleiche Studie aus dem Jahr 2025 belegt, dass mehr als 70 % der 2022 aufgefundenen Geheimnisse immer noch auftauchen und sich in Repositories, in denen GitHub Copilot aktiviert ist, 40 % mehr Geheimnisse befinden. [17]

Auch große Firmen wie Microsoft und Samsung sind vor diesen Risiken nicht sicher. Auch bei ihnen könnten Angreifer durch den geschickten Einsatz von Prompts Geheimnisse abgreifen, ohne hierfür Millionen von Codezeilen selbst analysieren zu müssen. Vergleiche [18].

Da LLMs wie Chat-GPT mit diesen öffentlich zugänglichen Daten trainiert werden, gelangen durch diese Daten auch die beschriebenen Risiken in die mit diesen Daten generierten Codes. Dagegen sind ein Großteil qualitativ hochwertiger Codes nicht öffentlich zugänglich und nur in firmeninternen Repositorien zu finden. In Appendix I ein einfaches Beispiel, welches eine mit OAuth 2.0 gesicherte API-Anfrage generieren lässt.

Im dargestellten Python-Script ist leicht zu erkennen, dass hier die Geheimnisse `CLIENT_ID` und `CLIENT_SECRET` beide im Quellcode auftauchen, in dem diese jedoch nichts zu suchen haben. Dies ist vor allem gefährlich für Nutzende, welche aufgrund fehlendem Wissen diese Probleme nicht erkennen und/oder nicht selbst beseitigen können. Hierdurch könnten so sensible Daten in Produktionsumgebungen verwendet oder öffentlich zugänglich gemacht werden. Diese potenziellen Schwachstellen bzw. Sicherheitslücken könnten Angreifer nutzen und als Einfallstore für Schadcodes verwenden. Andererseits ist es mit dem nötigen Fachwissen leicht möglich, diese Gefährdungen zu vermeiden, wenn sich Nutzende dieses Problems bewusst sind. Sie können dann durch einfaches Prompt Engineering das Problem vermeiden, wie beispielsweise in Appendix [J].

Zusätzlich werden LLMs auch an den Nutzerchats trainiert [I-C3]: “When you use our services for individuals such as ChatGPT, we may use your content to train our models.” [19] Hierdurch könnten von Chatnutzern eingegebene sensible Daten versehentlich veröffentlicht werden, ohne nachweisbar beispielsweise im Git Repository zu landen und so bei Sicherheitsprüfungen aufzufallen. [18]

IV. EINSATZPHASEN IM SOFTWAREENTWICKLUNGSPROZESS (SDLC)

Alle Formen der KI - speziell generative KI bzw. LLMs - können im Softwareentwicklungsprozess zum Einsatz kommen und erweitern regelmäßig die Möglichkeiten der Nutzenden. Besonders im Prototyping-Prozess von oft verwendeten Konzepten wie Web-Apps, in dem nur grobe Abläufe getestet werden sollen, beschleunigt die Verwendung von LLM diesen Prozess deutlich.

A. Planung

Der erste und zugleich oft wichtigste Schritt der Entwicklung von Software ist die Planung. [20] Hierbei muss das zu lösende Problem gründlich analysiert und danach alle Rahmenparameter definiert werden. Der richtige Einsatz von LLMs kann hier erfahrenen Entwicklern helfen, weitere Perspektiven oder Probleme einzubeziehen.

B. Design

Durch fehlendes Verständnis von Konzepten und darauf basierenden sehr häufig schlechten Designentscheidungen von LLMs erscheinen diese zur Verwendung im Designprozess nur wenig geeignet. "Machine Learning"-Prozesse können jedoch zur Analyse des Nutzerverhaltens verwendet werden und im Rahmen dieser Analyse zu Verbesserungen der Anwendung beitragen. [21]

C. Implementierung

Je nach Vorwissen der Nutzenden und dem Rahmen des Projektes sind LLMs in der Implementierung mehr oder weniger nützlich. Je weiter verbreitet die Problemstellung und je mehr öffentliche Daten dadurch zum Training der LLM verwendet werden können, desto effektiver werden die darauf basierenden Codevorschläge. So sind folgerichtig beispielsweise Vorschläge von LLMs für Web-Development deutlich besser, als die Vorschläge für Nischenprobleme wie die Lua-Entwicklung im Rahmen des Spieleframeworks von World of Warcraft (Blizzard Entertainment): "While LLMs achieve moderate performance on open-domain QA benchmarks, they are mainly successful on questions that probe knowledge that appears widely in their pre-training datasets." [22]

D. Testen

Ein essenzieller Teil der Softwareentwicklung ist das Erstellen von Tests. Leider werden Tests, die vollautomatisiert die Richtigkeit des geschriebenen Codes validieren, dennoch häufig, insbesondere wegen der niedrigen Priorität bei Kunden, bei dem Versuch Termine einzuhalten, als erstes gestrichen: "However, writing and maintaining high-quality test code is challenging and frequently considered of secondary importance." [23] LLMs können hier eingesetzt werden, den Aufwand des Testschreibens zu minimieren. Hierdurch könnten selbst bei einem gekürzten Zeitraum für das Test-Development immer noch sinnvolle Tests im Rahmen des hierdurch beschleunigten Prozesses erstellt werden. Wichtig ist jedoch, dies nicht zum Anlass zu nehmen, den für Tests allokierten Zeitrahmen des SDLC noch weiter zu kürzen.

E. Auslieferung & Wartung (CI-CD)

Nach Abschluss des Entwicklungsprozesses folgt die Auslieferung des Produktes. Da auch infolge rechtlicher Vorgaben ein Großteil der Produkte aktiv weiterentwickelt und gewartet werden muss, sollte großer Wert auf Continuous Integration und Continuous Development (CI-CD) gelegt werden. Diese beiden Begriffe stehen für die Automatisierung der Code-Integration und deren Validierung (CI) sowie die Automatisierung deren Auslieferung (CD) inklusive der Erstellung der notwendigen Datenbanken, serverseitiger Softwareupdates etc. Ergänzend zu den bisher manuell entwickelten Prozessen unterstützen nun auch KI gesteuerte Systeme die Entwickelnden. Eine im Jahr 2024 durchgeführte Studie untersuchte die

möglichen Vorteile von generativer KI in CI/CD. Sie kommt zu dem Ergebnis, dass diese die Möglichkeit beinhaltet, vor allem die Komplexität der Integration zu verringern und dadurch die Zufriedenheit von Nutzenden erhöhen kann: “[...] highlight the potential advantages of integrating GenAI, especially in terms of reducing integration complexity and increasing user satisfaction.” [24]

Als positives Beispiel sei hier auf Github hingewiesen, das Code Reviews durch GitHub Copilot anbietet, was Entwickelnden hilfreiches Feedback geben kann. Leider ist dieses Angebot jedoch nach aktuellem Stand je nach Framework oft noch fehlerhaft und muss dann manuell überprüft werden. Nachfolgend ein Beispiel aus dem Nischenkontext von WoW Lua. Siehe Fig 5:

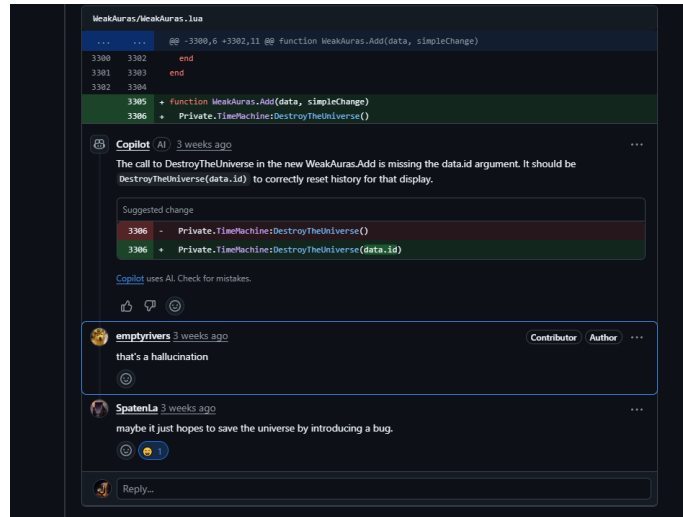


Fig. 5: Github Copilot
[25]

Zusammenfassend ist bei all diesen Möglichkeiten wichtig, LLMs nur als wertvolles Werkzeug einzusetzen und nicht als autonomen Ersteller von Lösungen zu verwenden: “Instead of letting an LLM take full control, they should behave like a consultant for the software engineers.” [26]

Folgerichtig kommt daher eine Studie aus dem April 2024 zu dem positiv zu wertenden Ergebnis, dass nur ca. 25 % der Entwickelnden der jeweils eingesetzten LLM blind vertrauen, wohingegen mehr als 33 % wenig bis gar kein Vertrauen in die Antworten der LLM haben. [27]

Selbst die Entwickler dieser Tools warnen vor blinden Vertrauen in die jeweils ausgegebene Lösung einer KI. So warnt beispielsweise Github bezüglich des mittels Copilot generierten Codes: “While Copilot Chat can be a powerful tool for generating code, it is important to use it as a tool rather than a replacement for human programming.” [28]

V. SICHERHEIT DER GENERIERUNG VON CODES DURCH LLMs

Viele Risiken und ungewollte Angriffsflächen von Software sind Entwickelnden häufig bekannt. Während des Schreibens der Software werden dennoch immer wieder - von Anfängern und Experten - Fehler übersehen. So sind auch heute noch hinlänglich bekannte Fehler wie Cross-Site-Scripting und SQL Injections zu finden. Auch hier könnten LLMs wie Copilot oder Codewhisperer die menschlichen Nutzer unterstützen, indem sie entweder vollautomatisch [29] oder - sicherer - nach Aufforderung [30] den Quellcode nach möglichen Schwachstellen durchsuchen und auf Lösungsvorschläge hinweisen.

Eine schon etwas ältere (2021) Studie zur Codesicherheit von Github Copilot ergab beispielsweise, dass knapp 40 % des generierten Codes Sicherheitslücken aufwies. [31]

Leider gab es hier in den letzten 4 Jahren, trotz all den Entwicklungen im Bereich der generativen KI, nur geringe Fortschritte. So erzeugte laut einer 2024 veröffentlichten Studie Chat-GPT immer noch durchschnittlich 15 % der Codes mit Kompilier- oder Runtime-Fehlern und es wurden nur 33 % des Codes ohne sicherheitsrelevante Probleme erkannt. [32] Diese Studie schloss jedoch auch, dass über 85 % der sicherheitsrelevanten Probleme durch entsprechendes Feedback und Multi-Interaktion mit der LLM gelöst werden konnten. Daher empfiehlt die Studie folgerichtig eine kritische Analyse zur Erkennung von Schwachstellen in LLM-basiertem Code. Eine weitere Studie [33] - spezifisch zur Programmiersprache C - fand in mehr als 51 % des Codes entsprechende Schwachstellen und stellte eine Korrelation dieser Schwachstellen zu den laut MITRE [34] 25 häufigsten Security-Schwachstellen fest.

Zusätzlich zur Verbesserung der Sicherheit durch regelhaftes Überprüfen des generierten Codes können auch durch das Zurückgreifen auf weniger allgemeine, auf spezifischen Datensätzen trainierte Language Modelle, sogenannte SLM [I-B], gute Ergebnisse erzielt werden. Diese haben deutlich weniger Parameter als LLMs, können jedoch durch ihr, auf ein Themengebiet fokussiertes Training gleichwohl oftmals bessere Antworten in ihrem Spezialgebiet geben. Vergleiche hierzu auch: [35]

VI. PRAKTISCHER NUTZEN VERSUS SICHERHEIT

Die Nutzung von generativer KI bietet zweifellos viele Vorteile wie verbesserte Qualität, Effizienz und eine, sich sogar bis zu 75 % [36] steigernde Beschleunigung mancher Entwicklungsprozesse. Nutzende sollten sich jedoch bei ihrem Einsatz stets der Risiken und spezifischen Probleme der Anwendung von generativer KI bewusst sein. Eine 2024 veröffentlichte Studie, welche Firmen und deren Nutzung von generativer KI untersucht hat, identifizierte Schlüsselherausforderungen, die bei der Verwendung im Softwareentwicklungsprozess, spezifisch der Generierung von Code, auftreten können und die daher besonders zu beachten sind:

A. *Datenschutz und Datensicherheit*

Durch die Nutzung von Daten innerhalb des Trainingsprozesses, insbesondere während des Trainings mittels Daten der Nutzerinteraktionen [I-C3], besteht regelmäßig die Gefahr, proprietäre Inhalte wie Quellcodes, sensible Nutzerdaten oder sogar strategische Firmeninformationen zu veröffentlichen [III]. Hierdurch können Rechte verletzt werden, die sowohl negative rechtliche, als auch finanzielle Folgen auslösen können.

B. *Faktisch inkorrekte Antworten*

Da LLMs nur mathematische Konstrukte sind, die mittels stochastischer Berechnungen anhand des Kontexts das nächste Wort erzeugen, passiert es häufig, dass dies auch zu faktisch inkorrekten Antworten führt. Im Entwicklungskontext einer Software kann dies zu ungelösten Problemen - technischen Schulden - führen, die jeweils bei Wartung der Software beseitigt werden müssen. Diese Altlasten wirken dann besonders schlimm, wenn Code durch sogenanntes “Vibe Coding” - einem Prozess, bei dem Nutzende den Code, den sie implementieren, nicht verstehen - implementiert wird. Durch das fehlende Code-Verständnis wird dann häufig nicht mehr wartungsfähiger Code erzeugt. Hier ist es häufig einfacher, diesen zu löschen und komplett neu zu erstellen, als zu versuchen, den vorher erzeugten Code zu verstehen und anzupassen.

C. *Halluzinationen*

Die Gefahr von Halluzinationen [II-D] sind im Softwareentwicklungsprozess besonders gravierend. Wenn LLMs Funktionen aufrufen, die nicht existieren, führt dies leider nur im Optimalfall direkt zu Fehlern, die schnell zu erkennen und zu beseitigen sind. Wesentlich schlimmer - vor allem bei Skriptsprachen - sind dadurch erzeugte Runtime-Fehler. Letztere sind insbesondere im Security-Kontext gefährlich, da sie für Angreifer regelmäßig einfache Angriffspunkte bieten.

D. *Ungewolltes Einbinden veralteter Versionen*

Neben der Gefahr von Halluzinationen führt das Verwenden von LLMs häufig zur Einbindung bereits veralteter Versionen von Bibliotheken und Funktionen [II-D]. Zu den allgemein bekanntesten Sicherheitshinweisen bei der Verwendung von elektronischen Geräten zählt nicht ohne Grund der Hinweis auf das zeitnahe und regelmäßige Aktualisieren der genutzten Software. Dies trifft auch auf die im Quellcode verwendeten Bibliotheken und Methoden zu. Nur dann können Sicherheitslücken direkt geschlossen und die Software gegen Angriffe ausreichend geschützt werden. Nutzende, die unwissentlich oder wegen mangelndem Sicherheitswissen dem, von der LLM generierten Quellcode vertrauen, können so bereits beseitigte Sicherheitslücken wieder implementieren und so neue Angriffspunkte erzeugen, die es Angreifern ermöglicht, Nutzerdaten abzugreifen oder ganze Systeme lahmzulegen. Vergleiche auch [37].

Zusätzlich zu diesen Problemen sind bei der Nutzung von LLMs natürlich auch ethische, Umwelt- und finanzielle sowie rechtliche Rahmenbedingungen zu beachten. Dabei ist es notwendig, jeweils die aktuellsten Normen und Vorgaben zu kennen, da nationale und internationale Gesetzgeber unter dem politischen Druck, Verbraucherschutz sicherzustellen, aktiv an der Regulierung von KI arbeiten. Beispielhaft sei hier auf den EU AI Act [38] sowie Novellen zur europäischen DSGVO [39] verwiesen.

VII. ZUSAMMENFASSENDE FESTSTELLUNGEN - CHANCEN UND HERAUSFORDERUNGEN GENERATIVER KI

Von der Anwendung in der Projektplanung bis hin zur Auslieferung und Wartung von Software bieten LLMs viele verschiedene Möglichkeiten, um den Softwareentwicklungsprozess zu erleichtern und zu beschleunigen. Anwender sollten sich daher stets der Herausforderungen bewusst sein, die die Anwendung im Alltag mit sich bringt: Angefangen beim Training durch Datensätze, welche bestimmte Gruppen oder Themen fehlrepräsentieren können [40], über die fehlende Nachvollziehbarkeit im Entscheidungsprozess von KI bis hin zur erforderlichen Sicherstellung eines ausreichenden Datenschutzes und der Datensicherheit sowie ethischer Herausforderungen, die die Anwendung von KI mit sich bringen kann. Hier bleibt es der Verantwortung der Nutzenden überlassen, dem sinnvoll entgegenzuwirken.

Aber nicht nur diese, auch bereits allgemein bekannte Herausforderungen sind für die korrekte Anwendung von KI relevant. Zusammengefasst ergeben sich nachfolgende Notwendigkeiten:

A. *Echokammer*

LLMs sind darauf trainiert, Nutzenden möglichst die Antwort zu geben, die diese hören wollen. Dies hat zur Folge, dass LLMs regelhaft häufiger Nutzenden zustimmen, als diesen zu widersprechen. Dies führt jedoch auch dazu, dass hierdurch Nutzende jeweils auch in ihren Überzeugungen verstärkt werden. [40] Darüber hinaus geben LLMs - außer sie wurden dahingehend angewiesen - nicht zu, in einem bestimmten Themengebiet nichts zu "wissen" und geben statt dessen - häufig glaubhaft aussehende - Antworten. Durch spezifisches Anpassen der Prompts sowie durch Anwenden von "Retrieval Augmented Generation (RAG)" [41], einer Methode, in der Prompts durch Dokumente - wie z.B. Firmenrichtlinien - mit entsprechenden Informationen angereichert werden, lässt sich dieses Risiko jedoch minimieren.

B. *Adaptivität*

Größere LLMs für bestimmte Themengebiete anzupassen, ist durch deren fehlende Generalisierungsfähigkeit und das fehlende Verständnis von Konzepten eingeschränkt. Durch das fehlende Verständnis für inhaltliche Themen ist eine LLM kaum in der Lage, sich an neue Aufgabenstellungen anzupassen. [42] Hierzu bietet sich alternativ die Verwendung von spezifisch auf die Themengebiete zugeschnittenen SLMs an.

C. *Kontextabhängigkeit*

Durch die Generierung von Texten mithilfe von Tokenwahrscheinlichkeiten [II-A] fehlt LLMs oft die Fähigkeit, längere Textpassagen einheitlich zu gestalten oder Gedankenstränge über längere Zeit fortzusetzen. [42] Dies kann dazu führen, dass die LLM - trotz kritischem Feedback - stets nur die gleichen Antworten abwechselt. Deutlich wird dies in folgendem Beispiel: Nutzende möchten von der LLM ein Script erstellt haben, das ein komplexes Problem löst. Das ausgegebene Produkt beinhaltet jedoch bereits die Fehler A und B. So kann es dazu kommen, dass die LLM nur einen der beiden Fehler beseitigt, aber gleichzeitig einen oder weitere Fehler einführt. Beim Beheben der neuen Fehler kann es dann geschehen, dass die alten, zwischenzeitlich bereits behobenen Fehler wieder eingeführt werden. Hier ist ein manuelles Eingreifen der Nutzenden unabdingbar, da nur so ein fehlerfreies Ergebnis erzeugt werden kann.

D. *Sich schnell ändernde Daten*

LLMs werden mit riesigen Datensätzen trainiert. Einige Themengebiete, insbesondere in der Softwareentwicklung, unterliegen jedoch massiven Entwicklungen und dadurch Veränderungen. Flutter, ein Framework zur Cross-Platform-App-Entwicklung, veröffentlicht beispielhaft im Dreimonatstakt Major-Versionen mit unzähligen Breaking-Changes. [43] Besonders gut dokumentierte Themengebiete leiden

daher alle unter dem Problem der sich stets verändernden Daten und sich zeitlich überholender Informationen. Bis dabei der kritische Satz an Informationen erreicht ist, orientiert sich die LLM an den veralteten Daten oder die Änderung erfolgt erst zu einem Zeitpunkt, an dem auch diese “neuen” Informationen bereits wieder veraltet sind. Nutzende müssen daher der Aktualität der Lösungen ständig erhöhte Aufmerksamkeit widmen.

E. Nachhaltigkeit

Das 2015 abgeschlossene und 2016 in Kraft getretene Pariser Klimaabkommen [44] hat auch Auswirkungen auf die Nutzung generativer KI. Der hohe Energieverbrauch von Modellen - sowohl während des Trainings als auch während der Nutzung - kann die Einhaltung der mit diesem Abkommen vereinbarten Ziele gefährden. Hier gilt es bereits bei der Planung durch Modellverbesserungen sowie Hardware-Upgrades und den Einsatz der jeweils richtigen Modelle und sorgfältiger Trainingsauswahl auf den möglichst geringen Energieverbrauch zu achten. Doch nicht nur die Anwendung von KI kann zu einem hohen Energieverbrauch führen. Es gilt auch darauf zu achten, dass generierte, zum Teil sogar qualitativ schlechtere Codes den Energieverbrauch nachteilig erhöhen können: Vergleiche. [45] Auch dies gilt es zu vermeiden.

F. Auswahl des richtigen Modells

Je nach Anforderung des Projektes bieten unterschiedliche Modelle unterschiedlich gute Unterstützung. Allgemeine Modelle - wie Chat-GPT, Gemini oder Lama - haben beispielsweise zwar mehr unterschiedliche Möglichkeiten, sind jedoch nicht immer die richtige Wahl. Die Auswahl des richtigen Modells sollte anhand der Requirements-Analyse IV-A ausgewählt werden. Vergleiche. [1]

G. Korrektheit des Codes

Auch syntaktisch korrekt generierter Code kann, obwohl er keinerlei Fehler produziert, im Laufe der Zeit zu logischen Problemen führen. Beispielfhaft sei dies bei folgendem, durch KI generiertem Primzahltest dargestellt:

```
def is_prime(n):
    if n < 2:
        return False
    # only check divisors up to sqrt(n)
    for i in range(2, int(n**0.5)):
        if n % i == 0:
            return False
    return True
```

Der Code überprüft - fehlerhaft - nur Werte von 1 bis Wurzel (n), ohne tatsächlich die Teilbarkeit durch die Wurzel zu überprüfen. Hierdurch werden Quadratzahlen fehlerhaft als Primzahlen erkannt.

```
print(is_prime(9))    # -> True    (9 = 3 x 3)
print(is_prime(25))   # -> True    (25 = 5 x 5)
print(is_prime(49))   # -> True    (49 = 7 x 7)
```

Die fehlerhaften Ausgaben erzeugen jedoch keinerlei Fehlermeldung und sind teilweise nur durch extensive Tests erkennbar. Hier sind die Nutzenden gefordert. [46]

H. Tatsächlicher Nutzen

Selbst bei Beachtung aller Notwendigkeiten bei der Nutzung von LLMs ist ihr Produktivitätsnutzen vor allem bei Senior-Softwareentwicklungen fraglich. Einen Prompt für eine komplexere Aufgabenstellung entsprechend korrekt vorzubereiten und entsprechend mit der LLM zu interagieren, ist oft so zeitaufwändig, dass es für erfahrene Nutzende häufig einfacher und schneller ist, eigenständig komplexere Probleme zu lösen. Eine Studie zur Steigerung der Produktivität von LLMs aus dem Open Source Bereich ergab folglich eine, von Entwickelnden geschätzte Steigerung der Produktivität bei ihrem Einsatz von 20 %. Tatsächlich sank jedoch die Produktivität um 19 % sank. [47] Fehlende Langzeitdaten machen es jedoch aktuell schwierig, die Nutzung von LLMs effektiv einzuschätzen. Doch bereits diese nur punktuellen Einsichten zeigen weitere Tücken der LLM-Nutzung im Rahmen der Softwareentwicklung auf.

Im Rahmen der Ausbildung von Softwareentwickelnden - in Theorie und Praxis - zeigt sich, dass häufig einfache, schon oft gelöste Fragestellungen erneut auftreten. Gleichwohl macht die Effektivität von LLMs, welche bei einfachen bis mittelschweren Aufgaben brillieren, deren Nutzung zu einer Versuchung, der nicht viele Nachwuchsentwickelnde widerstehen können. Dies führt leider durch die fehlende Auseinandersetzung mit dem technischen Basiswissen zu dann letztlich fehlender Sammlung von notwendiger Grundlagenerfahrung. Die durch die Nutzung von KI assistierten Tools so fehlende Erfahrung von Entwickelnden kann auf Dauer dann dazu führen, dass letztlich niedrig qualitativer Code einen immer größeren Stellenwert einnimmt.

APPENDIX A

REINFORCEMENT LEARNING CHAT-GPT

When given a Java program, first explain step-by-step what each part of the code does and what the overall goal is. Then, based on this reasoning, write a simple Python program that accomplishes the same task using idiomatic Python. Provide the reasoning clearly before showing the final Python code.

You're giving feedback on a new version of ChatGPT.

Which response do you prefer? Responses may take a moment to load.

Response 1

Thought for a few seconds >

Understood! Here's how I'll tackle any Java→Python translation you give me:

1. **High-level overview**
I'll start by summarizing what the Java program is supposed to do overall.
2. **Line-by-line / block-by-block walkthrough**
 - **Imports and package:** what libraries or packages it uses
 - **Class and method declarations:** what each class/method's purpose is
 - **Variables and data structures:** what data is being stored and why
 - **Control flow:** how loops, conditionals, exception handling, etc. work together
 - **I/O:** any input/output operations (console, files, network)
3. **Mapping to Python idioms**
For each Java construct, I'll explain how Python approaches the same task more succinctly or "Pythonically" (e.g. list comprehensions, duck typing, built-in functions).
4. **Step-by-step reasoning**
Before showing code, I'll walk you through my thought process:
 - How classes become modules or functions
 - How Java's static typing maps to dynamic Python types
 - How to handle exception logic, resource management, etc.
5. **Final Python code**
A clean, runnable script that mirrors the Java program's functionality in idiomatic Python.

Whenever you're ready, just paste in the Java code you'd like translated!

I prefer this response

Response 2

Thought for 13 seconds >

Here's the general recipe I'll follow any time you give me a Java program:

1. **High-Level Overview**
 - Identify the program's overall goal (e.g. "reads a file and counts words," "implements a simple calculator," "sorts an array," etc.).
2. **Structure Walk-Through**
I'll go through the Java code in the order you'd read it, explaining:
 1. **Package and imports** – what libraries or other code it depends on.
 2. **Class declaration** – the name of the class and any inheritance or interfaces.
 3. **Fields (variables) and constructors** – what state the class holds, and how it's initialized.
 4. **main method** – the entry point.
 5. **Other methods** – helper routines, their parameters, return types, and what they do.
 6. **Control flow** – loops, conditionals, exception handling.
 7. **I/O or side effects** – printing to console, reading files, network calls, etc.

At each step, I'll explain both **what** the code is doing and **why** (how it moves you toward the overall goal).

3. **Mapping to Python**
Once the Java behavior is crystal-clear, I'll rewrite it in Python, aiming for:
 - **Idiomatic structure:** use functions or, if needed, classes—but avoid Java-style boilerplate (no `public static void main`, no getters/setters unless really needed).
 - **Dynamic typing:** drop type declarations.
 - **Pythonic naming:** `snake_case` for functions/vars, `PascalCase` for classes.
 - **Built-ins:** prefer list comprehensions, `with`-blocks for I/O, exceptions directly, and the standard library.
 - **Main guard:** wrap script entry in `if __name__ == "__main__":`

Example Walk-Through

Suppose you hand me this little Java program:

Fig. 6: Reinforcement Learning

APPENDIX B

PROMPT ENGINEERING: ZEROSHOT

Give me an easy python program

Sure! Here's a simple Python program that asks the user for their name and age, then tells them how old they'll be in the next year:

```
python Kopieren Bearbeiten
```

```
# Simple Python Program: Greet and Predict Age

# Ask for user input
name = input("What's your name? ")
age = int(input("How old are you? "))

# Calculate age next year
next_year_age = age + 1

# Output a message
print(f"Hello, {name}!")
print(f"Next year, you'll be {next_year_age} years old.")
```

How to run it:

1. Copy the code into a `.py` file (e.g., `greet.py`).
2. Run it using a Python interpreter (e.g., `python greet.py`).

Want it to do something different or add a fun twist?

Fig. 7: Beispiel Zero Shot

APPENDIX C

PROMPT ENGINEERING: ONE & FEW SHOT

Give me a simple python program use this basic java programm as an example:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World");
    }
}
```

Sure! Here's a simple Python program that does the same thing as your Java example:

python

Kopieren

Bearbeiten

```
# This is a simple Python program
def main():
    print("Hello, World")

if __name__ == "__main__":
    main()
```

Explanation:

- `def main():` defines a function called `main`.
- `print("Hello, World")` prints the message.
- `if __name__ == "__main__":` ensures that `main()` is only called when the script is run directly, similar to `public static void main` in Java.

This is the Pythonic equivalent of your Java `HelloWorld` class.

Fig. 8: Beispiel One & Few Shot

APPENDIX D

PROMPT ENGINEERING: SYSTEM PROMPTING

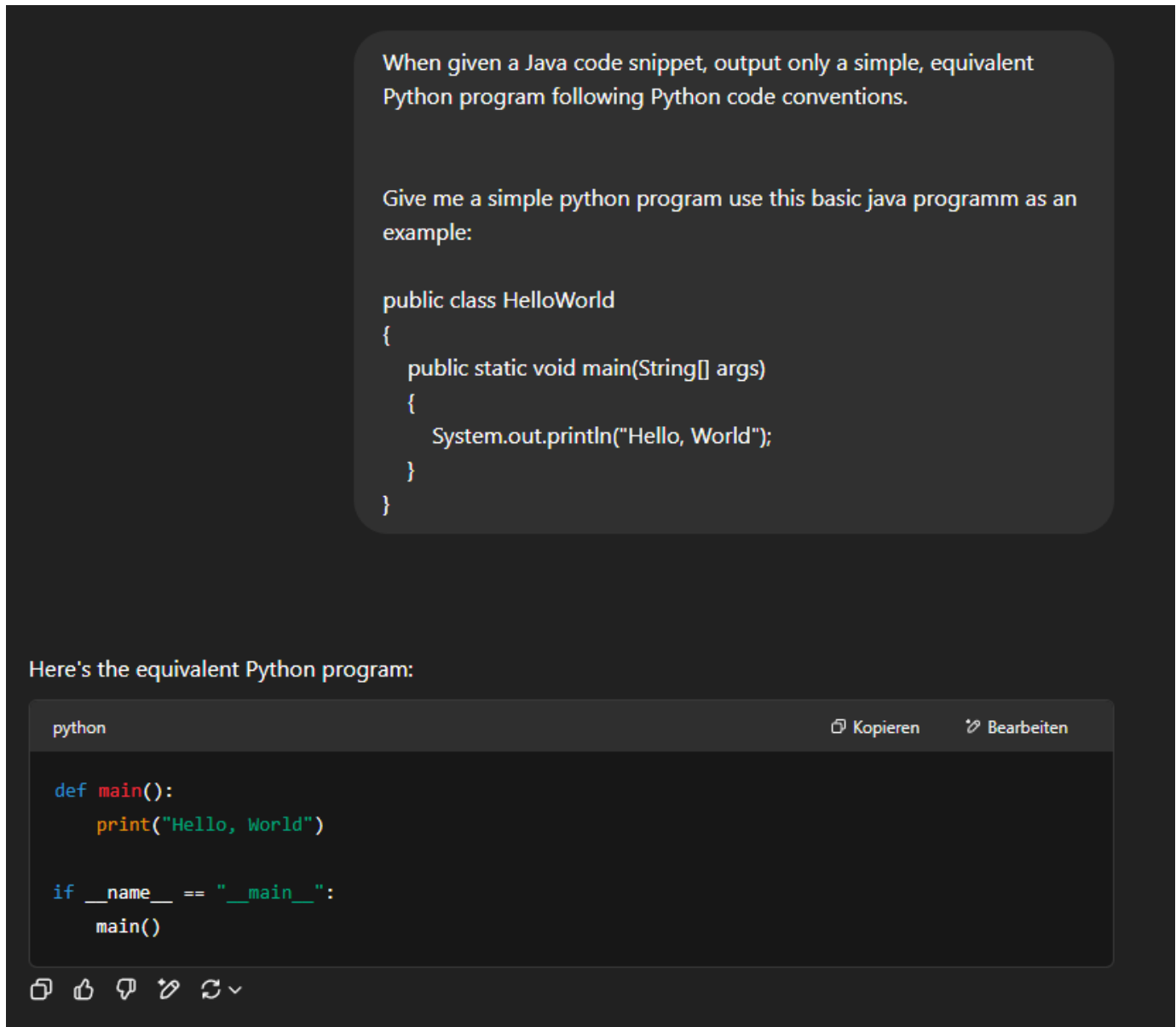


Fig. 9: Beispiel System Prompting

APPENDIX E

PROMPT ENGINEERING: CONTEXT PROMPTING

The following Java program prints "Hello, World" to the console:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World");
    }
}
```

Convert it into an equivalent Python script.

Here's the equivalent Python script for the given Java program:

```
python
```

Kopieren Bearbeiten

```
# This Python script prints "Hello, World" to the console

print("Hello, World")
```

Key Differences:

- No class or `main` method is needed in Python for a simple script.
- Python uses indentation instead of braces `{}`.
- `System.out.println` becomes just `print` in Python.

📄 👍 🗑️ ✎️ ↺

Fig. 10: Beispiel Context Prompting

APPENDIX F

PROMPT ENGINEERING: ROLE PROMPTING

You are a Pirate dedicated to translating code between the languages. Convert the following Java Code to python

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World");
    }
}
```


Arrr matey! Ye Java treasure be speakin' loud and proud, but I'll be translatin' it into the smooth Python tongue for ye:

python

Kopieren Bearbeiten

```
# Aye, this be Python now!
def main():
    print("Hello, World")

if __name__ == "__main__":
    main()
```

 **Explanation from the Code Docks:**

- Java's `public static void main(String[] args)` is swapped for Python's `main()` function.
- `System.out.println` be replaced with Python's simple `print`.
- The `if __name__ == "__main__"` line makes sure the ship only sails when the script be run directly.

Hoist the sails and sail on, ye code-be-speakin' swashbuckler!






    

Fig. 11: Beispiel Role Prompt

APPENDIX G

PROMPT ENGINEERING: STEP BACK PROMPTING

Before answering, reflect on the task: What is the underlying goal of converting a Java program to Python? Identify the key concepts being demonstrated in the Java code. Then, produce a simple and equivalent Python script that conveys the same idea using Python conventions.

Great! Let's walk through the task with careful reflection before diving into the code.

Reflection on the Task

1. Underlying Goal of Converting Java to Python:

The primary goal of converting a Java program to Python is not to directly translate syntax but to:

- Preserve the original logic and functionality, and
- Express it idiomatically in Python, using its simpler and more concise syntax.

This also involves shifting from object-heavy and type-explicit Java code to Python's more flexible and readable style.

If you'd like me to convert a specific Java snippet, feel free to share it and I'll walk through the same process.

🗑️ 📄 🔄 🌐 🏠

Translate this snippet:

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World");
    }
}
```

Let's apply the reflection process before translating the code.

Task Reflection

Goal:

Convert a simple Java program to Python in a way that retains the core purpose — displaying a greeting — while using idiomatic Python conventions.

Key Concepts in the Java Code

```
java
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World");
    }
}
```

This Java code demonstrates:

- A class definition (`HelloWorld`)
- A `main` method as the entry point
- Printing text to the console

Python Equivalent Using Pythonic Conventions

In Python, you don't need to define a class or a `main` method for such a simple script. Here's the idiomatic Python version:

```
python
print("Hello, World")
```

Summary

Concept	Java	Python
Class definition	Required (<code>public class ...</code>)	Not needed
Main method	<code>public static void main(String[] args)</code>	Not needed
Console output	<code>System.out.println(...)</code>	<code>print(...)</code>

Fig. 12: Beispiel Step Back Prompting

APPENDIX H

PROMPT ENGINEERING: CHAIN OF THOUGHT

When given a Java program, first explain step-by-step what each part of the code does and what the overall goal is. Then, based on this reasoning, write a simple Python program that accomplishes the same task. Provide the reasoning clearly before showing the final Python code.

```

public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World");
    }
}

```

Let's walk through the Java program piece by piece:

- `public class HelloWorld`
 - Declares a new class named `HelloWorld`.
 - In Java, every runnable program must have at least one `public` class whose name matches the filename (`HelloWorld.java`).
- `{ ... }` around the class
 - Defines the body of the class. Everything between these braces belongs to `HelloWorld`.
- `public static void main(String[] args)`
 - `public`: This method is accessible from outside the class.
 - `static`: You don't need to create an instance of `HelloWorld` to run this method—Java's runtime can call it directly.
 - `void`: The method does not return any value.
 - `main`: The entry point of any Java application; the JVM looks here first.
 - `String[] args`: An array of command-line arguments passed into the program (unused here).
- `System.out.println("Hello, World");`
 - `System.out`: Java's standard output stream (usually the console).
 - `println(...)`: Prints its argument, then terminates the line.
 - `"Hello, World"`: A string literal; exactly what will appear on the console.

Overall goal:
When you compile and run this program, it prints the text `Hello, World` to the console. It's the canonical "first program" in many languages, demonstrating basic structure, compilation, and output.

Translating into Python

Reasoning:

- Python does not require you to define classes or a special `main` method for a simple script.
- To print to the console, you use the built-in `print()` function.
- String literals work the same way (e.g. `"Hello, World"`).

Thus, the minimal Python equivalent simply invokes `print` with the same message:

```

python
print("Hello, World")

```

Fig. 13: Beispiel: Chain of Thought

APPENDIX I

GEHEIMNISSE IM CODE


Task:

Generate a Python script that connects to the Blizzard API (Battle.net), authenticates via OAuth2 client credentials, and fetches data from a chosen endpoint. The script should be well-structured, use best practices, and include clear comments.


Authentication:

Implement the OAuth2 "client_credentials" flow.


Here's a well-structured Python script that connects to the **Blizzard (Battle.net) API**, performs **OAuth2 authentication** using the `client_credentials` flow, and fetches data from a specified endpoint.

 **Prerequisites:**


- Python 3.7+
- `requests` library: Install via `pip install requests`

 **Script Overview:**

1. **Authentication** – Retrieves an access token using client credentials.
2. **API Request** – Uses the token to fetch data from a chosen Blizzard API endpoint.
3. **Best Practices** – Modular design, error handling, and comments.

 **Sample Use Case:**

Fetching World of Warcraft realms list from the Blizzard Game Data API.

 **Python Script:**

```
python                                                                    Kopieren Bearbeiten

import requests
import sys
import logging
from typing import Optional

# --- Configuration ---
BLIZZARD_CLIENT_ID = 'YOUR_CLIENT_ID'
BLIZZARD_CLIENT_SECRET = 'YOUR_CLIENT_SECRET'
```

Fig. 14: Beispiel: Geheimnisse im Code

APPENDIX J

GEHEIMNISSE NICHT IM CODE

Task: Generate a Python script that connects to the Blizzard API (Battle.net), authenticates via OAuth2 client credentials, and fetches data from a chosen endpoint. The script should be well-structured, use best practices, and include clear comments.


Authentication: Implement the OAuth2 "client_credentials" flow.

Security: Implement Security best practices e.g. no Secrets in sourcecode

Here's a well-structured and secure Python script that connects to the **Blizzard Battle.net API**, performs **OAuth2 client credentials** authentication, and fetches data from a specified endpoint. The script uses environment variables for storing sensitive credentials and follows best practices.

✓ Features:


- OAuth2 Client Credentials flow
- Uses environment variables for security
- Well-structured with modular functions
- Includes clear comments
- Handles errors and exceptions

 .env file (keep this file secret)

env

Kopieren Bearbeiten

```
BLIZZARD_CLIENT_ID=your_client_id_here
BLIZZARD_CLIENT_SECRET=your_client_secret_here
```

 fetch_blizzard_data.py

python

Kopieren Bearbeiten

```
import os
import requests
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

# Blizzard OAuth2 token URL
TOKEN_URL_TEMPLATE = "https://{region}.battle.net/oauth/token"

# Define environment variables
CLIENT_ID = os.getenv("BLIZZARD_CLIENT_ID")
CLIENT_SECRET = os.getenv("BLIZZARD_CLIENT_SECRET")
```

Fig. 15: Beispiel: Geheimnisse nicht im Code

APPENDIX K

HINWEISE ZU ”ILLUSION OF THINKING”

Wenige Tage vor Abgabeschluss der Arbeit wurde von Varela et al. ”Rethinking the Illusion of Thinking” [48] veröffentlicht, dass die Plausibilität von Illusion of Thinking in manchen Punkten anzweifelt: ”By introducing incremental stepwise prompting and agentic collaborative dialogue, we show that previously reported failures solving the Towers of Hanoi were not purely result of output constraints, but also partly a result of cognition limitations: LRMs still stumble when complexity rises moderately (around 8 disks). Moreover, the River Crossing results initially heralded as catastrophic failures turn out to hinge upon testing unsolvable configurations. Once we limit tests strictly to solvable problems—LRMs effortlessly solve large instances involving over 100 agent pairs.” [48] Leider konnten die Konsequenzen dieser Entdeckung beim Erstellen in der vorgelegten Arbeit nicht mehr eingearbeitet werden.

REFERENCES

- [1] Z. Örpek, B. Tural, and Z. Destan, ”The language model revolution: Llm and slm analysis,” in *2024 8th International Artificial Intelligence and Data Processing Symposium (IDAP)*, 2024, pp. 1–4.
- [2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, ”Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” *CoRR*, vol. abs/1712.01815, 2017. [Online]. Available: <http://arxiv.org/abs/1712.01815>
- [3] L. Ericsson, H. Gouk, C. C. Loy, and T. M. Hospedales, ”Self-supervised representation learning: Introduction, advances and challenges,” *CoRR*, vol. abs/2110.09327, 2021. [Online]. Available: <https://arxiv.org/abs/2110.09327>
- [4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, ”Efficient estimation of word representations in vector space,” 2013. [Online]. Available: <https://arxiv.org/abs/1301.3781>
- [5] J. Alammari. (2019) The illustrated word2vec. [Online]. Available: <https://jalammar.github.io/illustrated-word2vec/>
- [6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, ”Attention is all you need,” *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: <http://arxiv.org/abs/1706.03762>
- [7] (2025) Tokenizer gpt-4o. [Online]. Available: <https://platform.openai.com/tokenizer>
- [8] (2025) Wetter. [Online]. Available: https://www.duden.de/rechtschreibung/Wetter_Klima
- [9] (2025) Wettern. [Online]. Available: <https://www.duden.de/rechtschreibung/wettern>
- [10] O. H. Hamid, ”Beyond probabilities: Unveiling the delicate dance of large language models (llms) and ai-hallucination,” in *2024 IEEE Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA)*, 2024, pp. 85–90.
- [11] L. Boonstra. (2025) Prompt engineering. [Online]. Available: <https://www.kaggle.com/whitepaper-prompt-engineering>
- [12] SessionGloomy, ”Do anything now,” 2025. [Online]. Available: <https://www.reddit.com/r/ChatGPT/comments/10tevu1>
- [13] Y. Zeng, H. Lin, J. Zhang, D. Yang, R. Jia, and W. Shi, ”How johnny can persuade llms to jailbreak them: Rethinking persuasion to challenge AI safety by humanizing llms,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, L. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, 2024, pp. 14 322–14 350. [Online]. Available: <https://doi.org/10.18653/v1/2024.acl-long.773>
- [14] P. Shojaei*, I. Mirzadeh*, K. Alizadeh, M. Horton, S. Bengio, and M. Farajtabar, ”The illusion of thinking: Understanding the strengths and limitations of reasoning models via the lens of problem complexity,” 2025. [Online]. Available: <https://ml-site.cdn-apple.com/papers/the-illusion-of-thinking.pdf>
- [15] T. Godden. (2024) Your ai is only as good as your data. [Online]. Available: <https://aws.amazon.com/de/blogs/enterprise-strategy/your-ai-is-only-as-good-as-your-data/>
- [16] GitGuardian. (2023) The state of secrets sprawl 2023. [Online]. Available: <https://www.gitguardian.com/files/the-state-of-secrets-sprawl-report-2023>
- [17] —. (2025) The state of secrets sprawl 2025. [Online]. Available: <https://www.gitguardian.com/files/the-state-of-secrets-sprawl-report-2025>
- [18] Y. Huang, Y. Li, W. Wu, J. Zhang, and M. R. Lyu, ”Your code secret belongs to me: Neural code completion tools can memorize hard-coded credentials,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, p. 2515–2537, Jul. 2024. [Online]. Available: <http://dx.doi.org/10.1145/3660818>
- [19] (2025) What is chatgpt? [Online]. Available: <https://help.openai.com/en/articles/6783457-what-is-chatgpt>
- [20] (2025) Fixing the software requirements mess. [Online]. Available: <https://www.cio.com/article/255253/developer-fixing-the-software-requirements-mess.html>
- [21] C. Hymel, ”The ai-native software development lifecycle: A theoretical and practical new methodology,” 2024. [Online]. Available: <https://arxiv.org/abs/2408.03416>
- [22] N. Kandpal, H. Deng, A. Roberts, E. Wallace, and C. Raffel, ”Large language models struggle to learn long-tail knowledge,” 2023. [Online]. Available: <https://arxiv.org/abs/2211.08411>
- [23] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, ”When testing meets code review: Why and how developers review tests,” in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 677–687.
- [24] P. Sharma and M. S. Kulkarni, ”A study on unlocking the potential of different ai in continuous integration and continuous delivery (ci/cd),” in *2024 4th International Conference on Innovative Practices in Technology and Management (ICIPTM)*, 2024, pp. 1–6.
- [25] (2023) Copilot hallucination. [Online]. Available: https://github.com/WeakAuras/WeakAuras2/pull/4863#discussion_r2103247242

- [26] B. Wei, "Requirements are all you need: From requirements to code with llms," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, 2024, pp. 416–422.
- [27] R. Khojah, M. Mohamad, P. Leitner, and F. G. de Oliveira Neto, "Beyond code generation: An observational study of chatgpt usage in software engineering practice," 2024. [Online]. Available: <https://arxiv.org/abs/2404.14901>
- [28] (2025) Responsible use of github copilot chat in your ide. [Online]. Available: <https://docs.github.com/en/copilot/responsible-use-of-github-copilot-features/responsible-use-of-github-copilot-chat-in-your-ide>
- [29] t. b. S. K. Y. u. F. E. Steve Roberts, "Amazon codewhisperer," 2023. [Online]. Available: <https://aws.amazon.com/de/blogs/germany/amazon-codewhisperer-kostenlos-fuer-den-individuellen-gebrauch-ist-nun-allgemein-verfuegbar/>
- [30] (2025) Finding existing vulnerabilities in code. [Online]. Available: <https://docs.github.com/en/copilot/tutorials/copilot-chat-cookbook/security-analysis/finding-existing-vulnerabilities-in-code>
- [31] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," 2021. [Online]. Available: <https://arxiv.org/abs/2108.09293>
- [32] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No need to lift a finger anymore? assessing the quality of code generation by chatgpt," *IEEE Transactions on Software Engineering*, vol. 50, no. 6, pp. 1548–1584, 2024.
- [33] N. Tihanyi, T. Bisztray, R. Jain, M. A. Ferrag, L. C. Cordeiro, and V. Mavroeidis, "The formai dataset: Generative ai in software security through the lens of formal verification," in *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 33–43. [Online]. Available: <https://doi.org/10.1145/3617555.3617874>
- [34] "Cwe top 25 most dangerous software weaknesses," 2024. [Online]. Available: <https://cwe.mitre.org/top25/>
- [35] D. Dasgupta and A. Roy, "Issues with generic large language models (gllms)," in *2024 Artificial Intelligence for Business (AIB)*, 2024, pp. 47–50.
- [36] G. Paliwal, A. Donvir, P. Gujar, and S. Panyam, "Accelerating time-to-market: The role of generative ai in product development," in *2024 IEEE Colombian Conference on Communications and Computing (COLCOM)*, 2024, pp. 1–9.
- [37] M. Simaremare and H. Edison, "The state of generative ai adoption from software practitioners' perspective: An empirical study," in *2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2024, pp. 106–113.
- [38] Council of European Union, "Council regulation (EU) no 1689/2024," 2024. [Online]. Available: <https://eur-lex.europa.eu/eli/reg/2024/1689/oj>
- [39] Verordnung (EU) 2016/679, "Verordnung (eu) 2016/679 des europäischen parlaments und des rates vom 27. april 2016 zum schutz natürlicher personen bei der verarbeitung personenbezogener daten, zum freien datenverkehr und zur aufhebung der richtlinie 95/46/eg (datenschutz-grundverordnung) (text von bedeutung für den ewr)," 2016.
- [40] J. Nehring, A. Gabryszak, P. Jürgens, A. Burchardt, S. Schaffer, M. Spielkamp, and B. Stark, "Large language models are echo chambers," in *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*. Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING-2024), N. Calzolari, M.-Y. Kan, V. Hoste, A. Lenci, S. Sakti, and N. Xue, Eds. ELRA and ICCL, 2024.
- [41] B. Tural, Z. Örpek, and Z. Destan, "Retrieval-augmented generation (rag) and llm integration," in *2024 8th International Symposium on Innovative Approaches in Smart Technologies (ISAS)*, 2024, pp. 1–5.
- [42] D. H. Hagos, R. Battle, and D. B. Rawat, "Recent advances in generative ai and large language models: Current status, challenges, and perspectives," *IEEE Transactions on Artificial Intelligence*, vol. 5, no. 12, pp. 5873–5893, 2024.
- [43] "Flutter release notes," 2025. [Online]. Available: <https://docs.flutter.dev/release/release-notes>
- [44] UNFCCC, "Paris agreement." [Online]. Available: https://unfccc.int/sites/default/files/resource/parisagreement_publication.pdf
- [45] S. Sikand, R. Mehra, V. S. Sharma, V. Kaulgud, S. Podder, and A. P. Burden, "Do generative ai tools ensure green code? an investigative study," in *2024 IEEE/ACM International Workshop on Responsible AI Engineering (RAIE)*, 2024, pp. 52–55.
- [46] P. Preethi, V. Ragavan, C. Abinandhana, G. Umamaheswari, and D. Suvethaa, "Towards codeblizz: Developing an ai-driven ide plugin for real-time code suggestions, debugging, and learning assistance with generative ai and machine learning models," in *2024 International Conference on Emerging Research in Computational Science (ICERCS)*, 2024, pp. 1–7.
- [47] B. B. D. R. Joel Becker, Nate Rush, "Measuring the impact of early-2025 ai on experienced open-source developer productivity," 2025. [Online]. Available: https://metr.org/Early_2025_AI_Experienced_OS_Devs_Study.pdf
- [48] I. D. Varela, P. Romero-Soroazabal, E. Rocon, and M. Cebrian, "Rethinking the illusion of thinking," 2025. [Online]. Available: <https://arxiv.org/abs/2507.01231>