Universität Rostock

Traditio et Innovatio

Empirische Evaluation von LLM-generierten Code-Transformationen in realen Software-Systemen unterschiedlicher Komplexität

Name:	Ronald Nguyen
Matrikelnummer:	222200308
Abgabedatum:	19.03.2026
Betreuer und Gutachter:	Prof. Dr. Regina Hebig Universität Rostock Fakultät für Informatik
Gutachter:	Andreas Ruscheinski Universität Rostock Fakultät für Informatik

Bachelorarbeit am Institut für Informatik
Fakultät für Informatik und Elektrotechnik

Abstract

Diese Arbeit untersucht den Einfluss von Programmiersprache und Projektkomplexität auf die Qualität automatisierter Refactorings durch Large Language Modelle (LLMs). Ziel ist es, zu analysieren, inwieweit moderne LLMs in der Lage sind, verschiedene Refactoring-Aufgaben zuverlässig und korrekt umzusetzen. Hierfür wurden sechs Open-Source-Projekte aus den Programmiersprachen Python und Apex ausgewählt, die unterschiedliche Komplexitätsstufen aufweisen. In den Projekten wurden insgesamt sechs Refactoring-Typen definiert, die von einfachen Änderungen bis hin zu strukturell komplexeren Transformationen reichen. Zur Durchführung des Experiments wurden zwei LLMs eingesetzt: *Gemini 3 Pro* und *Codestral-2501*. Jedes Refactoring wurde pro LLM und Projekt zehnmal ausgeführt, um die nicht deterministische Natur von LLM-basierten Codegenerierungen zu berücksichtigen. Insgesamt wurden 720 Refactoring-Durchläufe durchgeführt. Die Qualität der Ergebnisse wurde anhand zweier Kriterien bewertet: der funktionalen Korrektheit, gemessen durch bestehende Regressionstests, sowie der korrekten Umsetzung des jeweiligen Refactorings mittels manueller Validierung. Die Ergebnisse zeigen deutliche Unterschiede zwischen den untersuchten Modellen. Während *Gemini 3 Pro* eine durchschnittliche Erfolgsrate von 94,7 % erreicht, liegt *Codestral-2501* mit 51,9 % deutlich darunter. Ein klarer Zusammenhang zwischen Projektkomplexität und Erfolgsrate konnte hingegen nicht festgestellt werden. Dagegen zeigen sich Unterschiede zwischen den Programmiersprachen: Insgesamt werden in Python-Projekten leicht höhere Erfolgsraten erzielt als in Apex-Projekten. Die Ergebnisse der Arbeit zeigen, dass moderne LLMs bereits in der Lage sind, einfache Refactorings zuverlässig umzusetzen, während komplexere strukturelle Transformationen weiterhin eine Herausforderung darstellen. Darüber hinaus verdeutlichen die Resultate, dass sowohl das verwendete Modell als auch der Refactoring-Typ einen stärkeren Einfluss auf die Erfolgsrate haben als die Komplexität der untersuchten Projekte. Der vollständige Programmcode sowie zusätzliche Ergebnistabellen sind im GitHub-Repository der Arbeit verfügbar [68] [1].

[1] https://github.com/Ronald-Nguyen/bachelor-thesis-llm-complexity]

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

List of Listings

Abkürzungsverzeichnis

1. Einleitung

Die vorliegende Arbeit untersucht den Einsatz von LLMs für die Durchführung von Refactorings in Softwareprojekten. Im Mittelpunkt steht die Frage, inwiefern sich die Qualität solcher Refactorings in Abhängigkeit von der Komplexität des Ausgangscodes und der verwendeten Programmiersprache unterscheidet. Dazu werden zunächst die Motivation und die Problemstellung erläutert, anschließend zentrale technische Grundlagen eingeführt sowie Zielsetzung, Forschungsfragen, methodisches Vorgehen und Aufbau der Arbeit dargestellt.

1.1. Motivation und Problemstellung

Minaee u. a. [64] zeigen, dass in den vergangenen Jahren beträchtliche Fortschritte bei LLMs erzielt worden sind. Insbesondere bei Aufgaben wie Codevervollständigung, automatisierter Fehlerbehebung und Codegenerierung erreichen aktuelle LLMs bereits ein Leistungsniveau, das in vielen Szenarien eine praktische Unterstützung für Entwicklerinnen und Entwickler darstellt. Dennoch ist bislang wenig darüber bekannt, wie differenziert diese Modelle in unterschiedlich komplexen Systemen Refactorings ausführen können. Es mangelt an systematischen Untersuchungen, die die Qualität der von LLMs ausgeführten Refactorings über Projekte unterschiedlicher Komplexität hinweg bewerten und deren Leistungsfähigkeit vergleichend einordnen. Zudem ist weitgehend ungeklärt, ob Refactorings in weniger verbreiteten Programmiersprachen wie Apex qualitativ von denen in etablierten Sprachen abweichen.

1.2. Technischer Hintergrund

In diesem Abschnitt werden die zentralen Begriffe dieser Arbeit kurz eingeführt. Eine vertiefte Darstellung der theoretischen Grundlagen erfolgt in Kapitel 2.

1.2.1. Refactoring

Wie Abid u. a. [2] definieren, ist das Refactoring eine Menge von Programmtransformationen, die darauf abzielen, das Systemdesign zu verbessern, ohne das äußere Verhalten zu verändern. Typische Ziele von Refactorings sind im Allgemeinen eine verbesserte Struktur, Lesbarkeit und Verständlichkeit von Code [39]. Besonders in großen Codebasen ist die Wartbarkeit entscheidend für langfristige Weiterentwicklung und Fehlerbehebung [39]. Vor allem in komplexem Code steigt für den Menschen das Risiko, unbeabsichtigt neue Fehler einzuführen [38]. Vor diesem Hintergrund ist manuelles Refactoring zeitaufwendig und anfällig für Fehler [38]. So stellt sich die Frage, inwiefern

LLMs den Menschen dabei ersetzen können, indem sie diese Aufgaben übernehmen. Code-Refactorings werden in dieser Arbeit als Basis verwendet, um eine systematische Bewertung der Qualität der von LLM generierten Ergebnisse vorzunehmen.

1.2.2. Softwaremetriken

Um die Komplexität eines Systems zu bestimmen und unterschiedliche Projekte vergleichbar zu machen, werden Softwaremetriken eingesetzt [37]. In dieser Arbeit spielen vorwiegend folgende drei Softwaremetriken eine zentrale Rolle:

- Cyclomatic Complexity (CC)
- Cognitive Complexity (COC)
- Lines of Code (LOC)

Die CC wird von McCabe [63] als „die Anzahl der linear unabhängigen Pfade im Kontrollflussgraphen eines Programms“ definiert. Veranschaulicht heißt das: Die CC eines Programms steigt mit der Anzahl seiner Verzweigungen und alternativen Pfade. Diese Kennzahl wurde ursprünglich hauptsächlich genutzt, um eine Schätzung darüber abzugeben, wie viele Testfälle nötig sind, um sämtliche Pfade im Code abzudecken [63]. Bis Dezember 2016 war es die einzige Metrik, um beurteilen zu können, wie schwer es ist, ein Programm zu verstehen, bis die COC eingeführt wurde [19]. Die COC wurde als Erweiterung zur CC eingeführt, um die Codeverständlichkeit statt die Testbarkeit zu prüfen. In Unterabschnitt 2.2.3 wird erläutert, weshalb die COC im Vergleich zur CC besser zur Bewertung der Verständlichkeit von Code geeignet ist. Die LOC geben an, wie viele Zeilen Quelltext ein Projekt umfasst. Diese Metrik ist zwar kein Qualitätsmaß, aber wichtig zur Einordnung der Projektgröße und zur Vergleichbarkeit der betrachteten Systeme [37]. In dieser Arbeit wird LOC insbesondere genutzt, um Projekte mit ähnlicher Größe, aber unterschiedlicher Komplexität gegenüberzustellen.

1.2.3. Apex

Apex ist eine objektorientierte Programmiersprache mit Java-ähnlicher Syntax, die konzeptionell an in Datenbanken gespeicherte Prozeduren angelehnt ist [96]. Sie ermöglicht es Entwicklerinnen und Entwicklern, Geschäftslogik direkt in Systemereignisse zu integrieren. Dazu zählen unter anderem Benutzerinteraktionen wie das Klicken auf Schaltflächen, die Aktualisierung verknüpfter Datensätze sowie die Verarbeitung von Visualforce-Seiten [96] [53]. Im Vergleich zu weitverbreiteten Programmiersprachen weist Apex eine deutlich geringere Verbreitung auf: Während auf *GitHub* [42] lediglich etwa 38.500 Repositorien existieren, die Apex verwenden, sind es bei Python rund fünf Millionen. Dies verdeutlicht die wesentlich größere Popularität und Community-Unterstützung von Python. Eine geringere öffentliche Codeverfügbarkeit könnte potenziell zu geringerer Repräsentation in Trainingsdaten führen. Daraus ergibt sich für den Einsatz von LLMs eine besondere Ausgangslage, da viele Modelle ihre Fähigkeiten aus großen Mengen öffentlich verfügbarer Trainingsdaten beziehen [18]. Die Qualität der Ergebnisse kann daher in weniger verbreiteten Sprachen variieren [20].

1.2.4. Large Language Modelle im Software Engineering

Heutzutage gibt es sehr viele LLMs welche eine große Rolle in der Softwareentwicklung spielen. Einige Modelle wurden speziell für Codegenerierungsaufgaben entwickelt, wie etwa *Codestral* [26], während andere als für Softwareentwicklungsaufgaben optimierte Varianten allgemeinerer Sprachmodelle bereitgestellt werden, wie *Codex* [70]. Andere, wie *Gemini 3 Pro*, eignen sich für allgemeine komplexe Aufgaben [33][64]. Es gibt bislang wenig systematische Erkenntnisse dazu, ob und wie stark die COC die Erfolgsrate und Qualität von LLM-Refactorings beeinflusst. Solche Erkenntnisse sind wichtig, um Szenarien zu identifizieren, in denen LLMs sinnvoll einsetzbar sind und in welchen nicht. Im Allgemeinen gibt es bereits viele Studien, welche Refactorings mit LLMs untersuchen [28]. Außerdem werden in der Praxis LLMs immer relevanter für Code-Generierung, da diese Modelle sich stetig verbessern [95].LLMs besitzen primär die Stärke in der schnellen Generierung von Vorschlägen und der Fähigkeit, Muster aus Trainingsdaten zu nutzen, welche optimale Attribute sind, um Refactorings auszuführen [15]. Dennoch ist zu berücksichtigen, dass die Verwendung von LLMs auch Nachteile mit sich bringt. Dazu zählen die Anfälligkeit für syntaktische und semantische Fehler sowie potenzielle Halluzinationen [58] [52]. Hinzu kommt der begrenzte Kontextumfang, der insbesondere bei großen Projekten problematisch wird [59]. Ferner liefern LLMs keine deterministischen Ergebnisse [18].

1.3. Zielsetzung und Forschungsfragen

Ziel der Arbeit ist es, empirisch zu untersuchen, inwiefern die COC und die Programmiersprache von bestehendem Code die Qualität von LLM-basierten Refactorings beeinflusst. Die Arbeit soll aufzeigen, inwieweit durch LLM-gestützte Refactorings die Ausführbarkeit und funktionale Korrektheit beeinflusst wird. Die Qualität der Ergebnisse wird über funktionale Korrektheit und Komplexitätsmetriken entschieden. Aus diesen Ergebnissen sollen die Chancen und Grenzen des Einsatzes von LLMs für strukturierten Refactorings abgeleitet werden. Zudem ist auch wenig bekannt, wie gut LLMs Refactorings mit unbekannteren Sprachen wie Apex im Vergleich zu bekannteren Sprachen wie Python ausgeführt werden können.

Folgende Forschungsfragen werden in dieser Arbeit adressiert:

- **RQ1:** Inwieweit beeinflusst die Cognitive Complexity die Codequalität von LLM-Refactorings?
- **RQ2:** Sind die von LLMs erzeugten Refactorings bei weniger verbreiteten Sprachen wie Apex qualitativ ebenso hochwertig wie bei etablierten Sprachen wie Python?

Hierbei wird die Qualität von LLM-basierten Refactorings nur dann als korrekt gewertet, wenn die Kombination aus funktionaler Korrektheit und der korrekte Implementierung des Refactorings besteht. Wobei die funktionale Korrektheit durch bestehende Tests gemessen wird, während die korrekte Implementierung von Refactorings manuell

ausgewertet wird. Die korrekte Implementierung wird anhand klar definierter Kriterien beurteilt.

1.4. Methodischer Ansatz (Überblick)

Um die Forschungsfrage beantworten zu können, gilt es Projekte zu suchen, welche den Anforderungen entsprechen. Für die erste Phase der Untersuchung wurden Python-Projekte ausgewählt. Python bietet sich unter anderem deshalb an, weil *GitHub* [42] eine sehr große Zahl an Open-Source Repositories besitzt und viele davon über eine Testinfrastruktur verfügen. Dadurch steht eine breite Auswahl an Projekten zur Verfügung, aus denen Beispiele mit geeigneter Größe und COC gewählt werden können. Für diese Arbeit wurden drei Python-Projekte und drei Apex-Projekte ausgewählt, die in Abschnitt 3.2 detailliert beschrieben werden. Die Projekte wurden so gewählt, dass sie eine vergleichbare Anzahl an LOC aufweisen, sich jedoch hinsichtlich ihrer COC unterscheiden. Auf diese Weise können Systeme mit niedriger, mittlerer und hoher COC gegenübergestellt werden, ohne dass Größenunterschiede die Ergebnisse verzerren. Somit ist der Einfluss der Komplexität gezielter beobachtbar. Außerdem wurden Refactorings definiert, welche die LLMs im anschließenden Verlauf des Forschungsexperiments auszuführen haben. Diese Refactorings wurden mithilfe von LLMs automatisiert 10-mal durchgeführt, um der nicht deterministischen Natur von LLMs vorzubeugen und zufällige Schwankungen in der Leistung auszugleichen. Die funktionale Evaluation erfolgt mittels Regressionstests. Die Umsetzung der Refactorings wird manuell beurteilt.

1.5. Aufbau der Arbeit

Kapitel 2 gibt einen Überblick über den technischen Hintergrund und zeigt, wie der aktuelle Stand der Forschung bezüglich LLMs und deren Anwendung im Software-Engineering ist. Der Schwerpunkt liegt dabei sowohl auf den relevanten Softwaremetriken als auch auf der Funktionsweise und Nutzung von LLMs in diesem Kontext.
Kapitel 3 erläutert die Methodik des Forschungsexperiments. Dabei wird dargestellt, nach welchen Kriterien die Projekte ausgewählt wurden. Außerdem wird erklärt, welche Rolle die eingesetzten Softwaremetriken spielen. Schließlich wird erläutert, wie sie angewendet werden. Zudem werden die Auswahl der Refactorings und der verwendeten LLMs beschrieben. Abschließend wird die technische Umsetzung des Forschungsexperiments erläutert.
Kapitel 4 präsentiert die Ergebnisse der Experimente und zeigt erste empirische Deutungen, welche aus den Ergebnissen ersichtlich sind.
Kapitel 5 diskutiert die Ergebnisse und vergleicht diese mit verwandten Arbeiten. Außerdem werden die Einschränkungen und Grenzen aufgezeigt sowie unerwartete Befunde und praktische Implikationen abgeleitet.
Kapitel 6 fasst abschließend die wichtigsten Erkenntnisse dieser Arbeit zusammen und gibt einen Ausblick auf die zukünftige Forschung.

2. Technischer Hintergrund und verwandte Arbeiten

In Kapitel 1 wurden die zentralen Begriffe dieser Arbeit bereits kurz eingeführt. In diesem Abschnitt wird tiefer auf den technischen Hintergrund eingegangen und zusätzlich werden verwandte Arbeiten aufgegriffen. Zunächst werden Refactorings systematisch eingeordnet. Anschließend werden Softwaremetriken betrachtet und verglichen. Darauf aufbauend wird der Einsatz von LLMs im Software-Engineering diskutiert. Abschließend werden verwandte Arbeiten zu LLM-basierter Codegenerierung und Refactoring zusammengefasst und in den Forschungskontext eingeordnet.

2.1. Refactoring

Refactorings bilden eine zentrale Grundlage dieser Arbeit. In den folgenden Unterkapiteln wird zunächst der Begriff definiert, unterschiedliche Arten von Refactorings eingeordnet und schließlich werden typische Probleme bei ihrer praktischen Durchführung dargestellt.

2.1.1. Begriffserklärung und Definition

Der Begriff Refactoring wurde Anfang der 1990er-Jahre von Opdyke und Johnson eingeführt und zunächst als Menge bedeutungserhaltender Umstrukturierungen für C++-Programme beschrieben [34]. Eine breite Etablierung in Forschung und Praxis erfuhr das Konzept insbesondere durch Fowler, der Refactoring als systematische Technik zur Verbesserung bestehenden Quellcodes popularisierte [39]. Aus Fowlers Arbeiten gingen insbesondere zwei Definitionen des Begriffs Refactoring hervor, die in der Literatur häufig verwendet werden [39, 61]:

- Refactoring bezeichnet eine gezielte Änderung der internen Struktur eines Softwaresystems mit dem Ziel, dessen Verständlichkeit und Wartbarkeit zu verbessern, ohne das äußere Verhalten zu verändern [39].
- Software-Refactoring beschreibt die übergeordnete Tätigkeit, bei der ein System durch die Anwendung einer oder mehrerer solcher Refactorings strukturell angepasst wird, unter Beibehaltung des externen Verhaltens [39, 61].

Beide Definitionen sind für diese Arbeit relevant und werden in der Praxis oft als Synonyme verwendet, wobei der Kontext meist klarstellt, ob eine einzelne Transformation oder ein umfassenderer Prozess gemeint ist. Unabhängig von den Begrifflichkeiten bleibt

das zentrale Prinzip gleich. Die funktionale Semantik des Codes muss erhalten bleiben, während dessen interne Qualität gezielt verbessert wird. Daraus ergeben sich insbesondere folgende Zielsetzungen [40]:

- Verbesserung der Lesbarkeit,
- Erhöhung der Wartbarkeit,
- klarere und konsistentere Strukturierung des Codes,
- Reduktion von Redundanzen.

Insbesondere in großen und historisch gewachsenen Codebasen ist eine hohe Wartbarkeit entscheidend für die langfristige Weiterentwicklung und zuverlässige Fehlerbehebung. Refactorings werden daher häufig in kleinen, inkrementellen Schritten durchgeführt, um das Risiko unbeabsichtigter Verhaltensänderungen zu minimieren und eine kontinuierliche Qualitätsverbesserung zu ermöglichen [61, 39].

2.1.2. Klassifikation von Refactorings

In der Literatur existieren verschiedene Ansätze zur Klassifikation von Refactorings. Fowler und Beck [39] ordnen Refactorings primär nach der Art der strukturellen Veränderung sowie nach dem jeweils verfolgten Ziel. Eine ergänzende Perspektive bietet die Einteilung nach dem Abstraktionsgrad der Transformation. In diesem Zusammenhang unterscheiden Abid u. a. [2] Refactorings zwischen der Code-, Design- und Architekturebene. Diese Klassifikation erlaubt eine differenziertere Betrachtung der Auswirkungen von Refactorings auf unterschiedlichen Ebenen eines Softwaresystems. Insbesondere im Kontext automatisierter Ansätze wurden zudem sogenannte suchbasierte Refactorings untersucht, bei denen Optimierungsverfahren eingesetzt werden, um geeignete Refactoring-Sequenzen in großen Suchräumen zu identifizieren [61]. Die Grundidee von suchbasierten Refactorings ist, dass Refactorings nicht direkt mit einer festen Regel ausgeführt, sondern wie ein Suchproblem behandelt werden und die unterschiedlichen Kombinationen möglicher Refactoring-Operationen zu dem großen Suchraum führen [61]. Diese unterschiedlichen Klassifikationen bilden die Grundlage für die Auswahl und Einordnung der in dieser Arbeit betrachteten Refactoring-Aufgaben.

2.1.3. Probleme mit Refactorings

Trotz ihrer zahlreichen Vorteile sind Refactorings in der praktischen Anwendung mit verschiedenen Herausforderungen verbunden. Manuelle Refactorings erfordern ein tiefes Verständnis der bestehenden Codebasis, da bereits geringfügige strukturelle Änderungen unbeabsichtigte Nebenwirkungen auf das Programmverhalten haben können [39]. Insbesondere in komplexen Softwaresystemen steigt das Risiko, durch Refactorings neue Fehler einzuführen, was den Aufwand für Tests und Validierung erheblich erhöht [2]. Darüber hinaus werden notwendige Refactorings in der Praxis häufig aus Zeitmangel oder aufgrund unzureichender Testabdeckung nur teilweise oder gar nicht durchgeführt, obwohl ein klarer Bedarf besteht. Dadurch verbleiben strukturelle Mängel im System,

was die Anhäufung technischer Schulden begünstigt und den zukünftigen Wartungs- und Änderungsaufwand erhöht [55]. Ein zentrales Problem besteht zudem darin sicherzustellen, dass Refactorings tatsächlich das bestehende Verhalten erhalten und keine unerwünschten Änderungen verursachen [39]. Diese Herausforderungen motivieren den Einsatz automatisierter Refactoring-Ansätze, insbesondere solcher auf Basis von LLMs, deren Potenzial und Grenzen im weiteren Verlauf dieser Arbeit untersucht werden.

2.2. Softwaremetriken

Dieser Abschnitt stellt die Grundlagen zu Softwaremetriken dar, anhand derer die betrachteten Projekte eingeordnet werden. Dazu gehören das Aufschlüsseln der CC und der COC sowie der Vergleich und die Abgrenzung beider Softwaremetriken.

2.2.1. Cyclomatic Complexity

Um Qualität und Komplexität von Quellcode systematisch bewerten zu können, werden in der Softwaretechnik seit mehreren Jahrzehnten Metriken eingesetzt, die eine objektive und reproduzierbare Einschätzung ermöglichen. Eine der ältesten und am weitesten verbreiteten Metriken ist die Cyclomatic Complexity, die ursprünglich von McCabe [63] eingeführt wurde und die strukturelle Verzweigungsvielfalt eines Programms über die Anzahl linear unabhängiger Kontrollflusspfade erfasst. In der Praxis wird die CC häufig als Heuristik für erhöhten Testaufwand verwendet, da mit steigender Anzahl unabhängiger Pfade typischerweise mehr Testfälle erforderlich werden, um die unterschiedlichen Ausführungsfälle abzudecken [93]. Gleichzeitig wird in empirischen Arbeiten diskutiert, dass CC stark mit Größenmaßen (z.B. LOC) zusammenhängen kann und deshalb als alleinige Qualitäts- bzw. Verständlichkeitsmetrik nur begrenzt trennscharf ist [48].

Listing 2.1.: Beispiel zur Berechnung der Cyclomatic Complexity

```
1  foo = 10
2  bar = 5
3
4  if foo > 0:                    # Entscheidung 1
5      if bar > foo:              # Entscheidung 2
6          print("bar > foo")
7      else:
8          print("bar <= foo")
9  else:
10     if bar < 0:                # Entscheidung 3
11         print("bar < 0")
12     else:
13         print("bar >= 0")
```

Zur Veranschaulichung zeigt Listing 2.1 ein Codebeispiel mit drei Entscheidungspunkten. Die Berechnung der CC erfolgt ausgehend von einem Basiswert von eins (linearer

Kontrollfluss); jeder zusätzliche Entscheidungspunkt, der alternative Ausführungspfade erzeugt, erhöht die Metrik um eins [63, 46]. Für das Beispiel ergibt sich damit:

$$\mathrm{CC} = 1 + 3 = 4.$$

Die Interpretation dieser Zahl ist primär strukturell: Sie beschreibt die Anzahl logisch unabhängiger Pfade und motiviert damit insbesondere testbezogene Überlegungen; als direkte Metrik menschlicher Verständlichkeit ist CC dagegen nur eingeschränkt geeignet [19] [66] [37] [48].

2.2.2. Cognitive Complexity

Aufgrund der beschriebenen Einschränkungen klassischer, stark strukturbezogener Metriken wurde mit der Cognitive Complexity eine Metrik vorgeschlagen, die explizit auf Aspekte der menschlichen Codeverständlichkeit abzielt. Campbell [19] beschreibt COC als Metrik, die nicht nur Entscheidungspunkte zählt, sondern insbesondere Verschachtelungen, Unterbrechungen des linearen Leseflusses sowie zusätzliche mentale Kontextwechsel berücksichtigt. Dadurch erhalten semantisch komplexere Kontrollstrukturen typischerweise höhere Werte als bei reiner Pfad-/Entscheidungszählung. Empirische Validierungen zeigen zudem, dass COC mit Messgrößen der Verständlichkeit (z.B. benötigte Verständniszeit oder subjektive Ratings) korreliert und damit zumindest Teilaspekte von „Understandability“ sehr gut reflektieren kann [66].

Listing 2.2.: Beispiel zur Berechnung der Cognitive Complexity

```
1  foo = 10
2  bar = 5
3
4  if foo > 0:                  # +1
5      if bar > foo:            # +1 +1 (verschachtelt)
6          print("bar > foo")
7      else:
8          print("bar <= foo")
9  else:
10     if bar < 0:              # +1 +1 (verschachtelt)
11         print("bar < 0")
12     else:
13         print("bar >= 0")
```

Im Beispiel in Listing 2.2 trägt jede Kontrollstruktur zunächst +1 bei. Zusätzlich erhöht die Verschachtelung innerer Kontrollstrukturen die COC. Damit kann derselbe Code, der in CC „nur“ als Menge von Entscheidungspunkten erscheint, in COC stärker gewichtet werden, sobald tiefe Verschachtelungen oder mehrfach unterbrochene Kontrollflüsse vorliegen. Für das Beispiel ergibt sich damit:

$$\mathrm{COC} = 1 + 2 + 2 = 5.$$

2.2.3. Vergleich und Abgrenzung

Die CC und die COC adressieren unterschiedliche Zielgrößen und sollten daher als komplementär verstanden werden: CC ist primär eine strukturorientierte Metrik mit engem Bezug zu Pfaden und damit zu testbezogenen Überlegungen [63, 93], während COC gezielt Merkmale modelliert, die beim Lesen von Code kognitive Zusatzlast erzeugen (z.B. Verschachtelung und Kontextwechsel) [19]. Entsprechend kann Code mit moderater CC eine hohe COC aufweisen, wenn die Logik schwer nachzuverfolgen ist, während eine erhöhte CC nicht automatisch geringe Verständlichkeit bedeutet, sofern die Struktur flach und linear lesbar bleibt. Für Refactorings, die explizit auf Lesbarkeit und Wartbarkeit zielen, ist COC deshalb häufig die passendere Evaluationsmetrik. Empirische Untersuchungen deuten zudem darauf hin, dass COC die wahrgenommene Verständlichkeit teils besser vorhersagt als CC [56, 66].

2.3. Large Language Modelle im Kontext des Software Engineering

Da LLMs inzwischen in vielen Bereichen des Software Engineering eingesetzt werden, ist ihre Einordnung für diese Arbeit von besonderer Bedeutung. Der folgende Abschnitt gibt daher einen Überblick über zentrale Einsatzfelder, bevor Potenziale und Grenzen dieser Modelle im softwaretechnischen Kontext betrachtet werden.

2.3.1. Grundlagen und Entwicklung von Large Language Modellen

LLMs haben in den letzten Jahren massive Fortschritte gemacht. Durch das Training mit umfangreichen Textsammlungen sind tiefe neuronale Netze in der Lage, komplexe Aufgaben mittels Mustererkennung zu bewältigen [64]. Transformerbasierte Architekturen stellten einen Wendepunkt dar, weil sie langfristige Abhängigkeiten effizient modellieren. Sie ermöglichen eine vollständig parallele Verarbeitung von Daten und können Abhängigkeiten im Text effektiv erfassen [92]. Diese Architektur ist in jedem LLM enthalten [95].

2.3.2. LLM-basierte Codegenerierung

Die automatisierte Codegenerierung gehört zu den wichtigsten Anwendungsfeldern von LLMs im Software Engineering. Mit Codex wurde ein speziell auf Quellcode trainiertes Sprachmodell untersucht, das funktionsfähigen Code aus natürlichsprachlichen Beschreibungen erzeugen kann [22]. Zur Evaluation wurde unter anderem *HumanEval* verwendet, ein von OpenAI entwickelter Benchmark, der in der Forschung weit verbreitet ist und zur Bewertung der funktionalen Korrektheit generierter Programme dient [22]. Der Benchmark umfasst 164 sorgfältig konstruierte Programmieraufgaben, die jeweils durch automatisierte Tests überprüft werden. Ein zentrales Evaluationsmaß ist dabei `pass@k`. Die Kennzahl `pass@1` gibt an, mit welcher Wahrscheinlichkeit mindestens eine von genau einer generierten Lösung die zugehörigen Testfälle vollständig besteht

[22]. Sie misst somit die Erfolgsrate des ersten Modellvorschlags ohne Mehrfachversuche oder Auswahlmechanismen. Beispielsweise erreichte das LLM *Codestral-2501* bei den Python-Aufgaben im `pass@1`-Setting eine Erfolgsquote von 86,6 % [26]. Höhere k-Werte wie `pass@10` berücksichtigen mehrere generierte Kandidaten und erhöhen entsprechend die Wahrscheinlichkeit, dass mindestens eine Lösung korrekt ist. `Pass@k` wird in [22] formal wie folgt definiert:

$$\text{pass@}k := \mathbb{E}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right].$$

Dabei bezeichnet:

- n die Gesamtanzahl generierter Lösungskandidaten für eine Aufgabe
- c die Anzahl korrekter Lösungen unter diesen Kandidaten
- k die Anzahl betrachteter Stichproben

Der Ausdruck

$$\frac{\binom{n-c}{k}}{\binom{n}{k}}$$

beschreibt somit die Wahrscheinlichkeit, dass bei einer Auswahl von k Lösungen ausschließlich fehlerhafte Kandidaten gezogen werden. Durch Subtraktion von eins wird die Gegenwahrscheinlichkeit berechnet, also die Wahrscheinlichkeit, dass mindestens eine der k ausgewählten Lösungen korrekt ist[64].

2.3.3. Herausforderungen, Limitationen und Risiken

Trotz ihrer Leistungsfähigkeit haben LLMs grundlegende Schwächen. Das zentrale Problem sind Halluzinationen. Modelle können Code erzeugen, der syntaktisch plausibel aussieht, aber inhaltlich falsch ist. Halluzinationen sind demnach ein typisches Verhalten von LLMs [52]. Generierter Code kann also nicht ohne weitere Validierung als korrekt oder sicher angesehen werden. Speziell in LLM-generiertem Code treten selbst bei scheinbar funktional korrekten Programmen subtile semantische Fehler auf [58]. Dazu kommt, dass LLMs keine deterministischen Ergebnisse liefern, was die Reproduzierbarkeit erschwert [18]. Der begrenzte Kontextumfang führt besonders bei größeren Codebasen zu unvollständigem Programmverständnis [64].

2.4. Verwandte Arbeiten

Um die vorliegende Arbeit in den bestehenden Forschungskontext einzuordnen, werden in diesem Abschnitt relevante verwandte Arbeiten vorgestellt. Der Fokus liegt auf zentralen Studien zu LLM-basierten Refactorings, deren Evaluation sowie möglichen sprachabhängigen Unterschieden. Diese Arbeiten werden im Hinblick auf ihre Bedeutung für die eigene Untersuchung eingeordnet und von dieser abgegrenzt.

2.4.1. Empirische Arbeiten zu LLM-basierten Refactorings

Die Forschung zu LLM-basierten Refactorings hat in den letzten Jahren deutlich zugenommen [64]. Mehrere Arbeiten untersuchen, ob und wie gut LLMs in der Lage sind, bestehenden Code zu verbessern, ohne dessen Verhalten zu ändern [81, 24, 28]. Shirafuji u. a. [81] zeigen, dass *Few-Shot-Prompting* bei *GPT-3.5* die CC von Python-Code durchschnittlich um 17,35% reduzieren kann, wenn nur die semantisch korrekten generierten Programme berücksichtigt werden. *Few-Shot-Prompting* bezeichnet eine Methode, bei der dem Modell neben der eigentlichen Eingabe zusätzlich eine kleine Anzahl von Beispielen im Prompt bereitgestellt wird, um das gewünschte Verhalten zu steuern [18]. In der Untersuchung erreichte das *Few-Shot-Prompting* eine Erfolgsquote von 91,11% bei `pass@1`. Damit übertraf die Methode sowohl das *Zero-Shot-Prompting* (87,44%) als auch das *One-Shot-Prompting* (89,77%). *Zero-Shot-Prompting* erfolgt ohne zusätzliche Beispiele im Prompt, während beim *One-Shot-Prompting* genau ein Beispiel zur Steuerung des Modellverhaltens bereitgestellt wird [79]. Bei `pass@10` lag der Wert bei 95,68%. Die Arbeit zeigt auch, dass LLMs gut in der Code-Formatierung sind, jedoch teilweise unerwünschte Verhaltensweisen wie das Löschen von Kommentaren aufweisen. Eine ähnliche Richtung verfolgen Choi, An und Yoo [24], die einen iterativen Ansatz zur Reduktion der CC in realen Java-Open-Source-Projekten aus *Defects4J* untersuchen. *Defects4J* ist ein Datensatz mit realen Fehlern, also tatsächlich in der Praxis aufgetretenen und behobenen Bugs in Java-Projekten [51]. Bei über 20 Iterationen erreichen sie eine Reduktion der durchschnittlichen CC um bis zu 10,4%. Ihr Vorgehen identifiziert automatisch Methoden mit hoher Komplexität und lässt diese gezielt durch ein LLM umstrukturieren. Die Ergebnisse zeigen, dass projektweites Refactoring mit LLMs machbar ist, die Effektivität aber von der Projektgröße und der Codestruktur abhängt. Cordeiro, Noei und Zou [28] vergleichen *StarCoder2* mit menschlichen Entwicklern über 30 Java-Projekte hinweg. Das Modell reduziert Code Smells um 20,1% mehr als Entwickler und zeigt Stärken bei systematischen Problemen wie Long Statement und Magic Number. Bei kontextabhängigen Problemen schneiden Menschen besser ab. *One-Shot-Prompting* erhöht die Unit-Test-Pass-Rate um 6,15% und die Code-Smell-Reduktion um 3,52%. Diese Arbeit zeigt, dass die Kombination aus One-Shot-Prompting und mehreren generierten Refactorings die beste Performance liefert. Eine weitere Studie deutet darauf hin, dass weder LLMs noch klassische statische Analysewerkzeuge bei der Erkennung von Code Smells durchgängig überlegen sind, sondern je nach Code Smell-Typ unterschiedliche Stärken zeigen [84]. Eine breite Übersicht bietet die systematische Literaturübersicht von Martinez u. a. [62]. Sie analysieren 50 Primärstudien und kategorisieren diese nach Refactoring-Methoden, Prompt-Engineering-Techniken, verwendeten LLM-Tools, Programmiersprachen und Datensätzen. Die Autoren identifizieren die *Extract Method* als häufigste Refactoring-Art mit vielversprechenden Ergebnissen. Gleichzeitig weisen sie darauf hin, dass LLMs oft fehlerhaften Code erzeugen, Schwierigkeiten mit komplexeren Refactorings haben und Anfragen von Entwicklern häufig missverstehen. Die Definition von Accuracy variiert stark zwischen den Studien, was einen standardisierten Vergleich erschwert. Cordeiro, Noei und Zou [29] diskutieren grundsätzliche Limitationen von LLM-basierten Refactorings. Sie betonen, dass die Fehleranfälligkeit und Halluzinationen von LLMs deren vollständige Integration in automatisierte Refactoring-Pipelines

(z.B. in Entwicklungsumgebungen) einschränken. Oft wird deshalb ein Mensch benötigt, um die durchgeführten Änderungen zu verifizieren. Die Autoren schlagen Forschungsrichtungen vor, um die Qualität LLM-basierter Refactorings zu verbessern und deren Zuverlässigkeit in der Softwarewartung zu erhöhen.

2.4.2. Evaluation und Qualitätsmetriken in verwandten Arbeiten

Die Bewertung von LLM-generierten Refactorings erfolgt in der Literatur häufig anhand der CC, um die strukturelle Komplexität vor und nach dem Refactoring zu vergleichen [81, 24, 28]. Wie in Unterabschnitt 2.2.3 diskutiert, erfasst CC primär die Anzahl unabhängiger Kontrollflusspfade und eignet sich damit für testbezogene Betrachtungen, während COC Verschachtelung und kognitive Last stärker gewichtet und deshalb näher an der wahrgenommenen Verständlichkeit liegt. Zusätzlich zu diesen Strukturmetriken wird in vielen Arbeiten die funktionale Korrektheit durch bestehende Unit-Tests geprüft [28, 81]. Einige Studien ergänzen die Evaluation durch die Identifikation von *Code Smells* vor und nach dem Refactoring [28]. *Code Smells* wie *Long Method*, *Large Class* oder *Magic Number* werden gezählt und deren Reduktion als Qualitätsgewinn interpretiert. Diese Metrik ist allerdings weniger formalisiert als CC oder COC und hängt stark von der verwendeten Methode ab. Ein weiteres Problem ist die fehlende Standardisierung bei der Definition von Korrektheit. Martinez u. a. [62] weisen darauf hin, dass die Bedeutung von Accuracy je nach Studie variiert. Manche Arbeiten prüfen nur syntaktische Korrektheit, andere zusätzlich semantische Äquivalenz, und wieder andere verlassen sich ausschließlich auf Tests [62]. Diese Unterschiede erschweren den direkten Vergleich zwischen Studien und unterstreichen die Notwendigkeit einheitlicher Evaluationsprotokolle. Trotz dieser Herausforderungen zeigen die vorhandenen Ergebnisse, dass LLMs grundsätzlich in der Lage sind, strukturelle Verbesserungen zu erzielen, die messbar sind [81]. Die Frage, wie zuverlässig und reproduzierbar diese Verbesserungen ausfallen, bleibt allerdings teilweise offen.

2.4.3. Sprachabhängigkeit von LLM-generierten Refactorings (RQ2)

Polyglotte Benchmarks wie *MultiPL-E* [20] und *Aider Polyglot* [3] zeigen, dass die Leistungsfähigkeit von LLMs stark von der Programmiersprache abhängt. RQ2 fragt gezielt danach, wie gut moderne LLMs Refactorings in weniger verbreiteten Sprachen wie Apex im Vergleich zu etablierten Sprachen wie Python umsetzen können. Diese Benchmarks evaluieren zwar die generellen Programmierfähigkeiten in vielen Sprachen, untersuchen aber nicht spezifisch Refactoring-Aufgaben. Besonders auffällig: Apex, eine für Salesforce-Entwicklung wichtige, jedoch nicht weit verbreitete Sprache, findet in keiner der etablierten Benchmark-Studien Berücksichtigung. Damit bleibt die Fähigkeit von LLMs, strukturierte Code-Transformationen in solchen Sprachen korrekt auszuführen, empirisch weitgehend unerforscht. Die vorliegende Arbeit schließt diese Lücke, indem sie erstmals die Refactoring-Qualität von *Gemini 3 Pro* und *Codestral-2501* systematisch zwischen Python- und Apex-Projekten vergleicht.

2.4.4. Abgrenzung der vorliegenden Arbeit

Im Gegensatz zu bisherigen Studien, die meist einzelne LLM wie *GPT-3.5*, *StarCoder2* oder *GPT-4* untersuchen [81, 28], vergleicht diese Arbeit direkt *Gemini 3 Pro* mit *Codestral-2501*. So wird erstmals klar, welches Modell für welche Refactoring-Aufgaben tatsächlich besser geeignet ist. Besonders charakteristisch ist der Fokus auf COC als zentrale Komplexitätsmetrik. Während verwandte Arbeiten primär CC oder Code Smells einsetzen, misst diese Arbeit gezielt die kognitive Verständlichkeit, also COC genau das, was Refactorings verbessern sollen. Die funktionale Korrektheit wird durch bestehende Tests geprüft, was praxisnahe Validierung ermöglicht. Hervorzuheben ist darüber hinaus der verwendete Datensatz: Anstelle gängiger Standard-Benchmarks wie *HumanEval* [81, 24] werden in dieser Arbeit sechs reale Open-Source-Projekte herangezogen, drei Python-Projekte mit kontrollierter COC-Varianz, sowie erstmals drei Apex-Projekte. Zusammengefasst untersucht diese Arbeit erstmals systematisch, wie stark COC und Programmiersprache (Python vs. Apex) die Qualität von LLM-Refactorings beeinflussen.

3. Methodik

In diesem Kapitel wird das methodische Vorgehen der Arbeit beschrieben. Zunächst wird das Forschungsdesign vorgestellt, das den Aufbau des Experiments sowie die Auswahl der Datensätze, Refactorings und LLMs umfasst. Anschließend wird das Prompt Engineering erläutert. Danach wird dargelegt, wie die erzeugten Refactorings validiert wurden und welche technische Umsetzung dem automatisierten Ablauf des Experiments zugrunde liegt. Auf diese Weise soll das Vorgehen der Arbeit transparent und nachvollziehbar dargestellt werden.

3.1. Forschungsdesign

In diesem Abschnitt wird das Forschungsdesign der vorliegenden Arbeit vorgestellt. Zunächst erfolgt ein Überblick über den Aufbau des Experiments. Anschließend werden die Auswahl der Projekte, die verwendeten LLMs sowie die Bewertung der Refactorings beschrieben.

3.1.1. Überblick über das Experiment

Die vorliegende Arbeit folgt einem kontrollierten und quantitativen Forschungsdesign. Ziel ist es, empirisch zu untersuchen, inwiefern die COC sowie die Programmiersprache einen Einfluss auf die Qualität von LLM-basierten Refactorings haben. Dabei werden LLMs gezielt auf ausgewählte Refactoring-Aufgaben angewendet und die resultierenden Veränderungen systematisch gemessen. Zur Beantwortung der ersten Forschungsfrage (RQ1) wurde die COC als zentrale unabhängige Variable verwendet. Im Fokus steht dabei die Frage, ob LLMs bei steigender COC des Ausgangscodes, Unterschiede in der Refactoring-Qualität zeigen. Die CC wurde in diesem Zusammenhang bewusst nicht berücksichtigt, da sie primär strukturelle Kontrollflussaspekte abbildet und nicht die für diese Arbeit relevante Verständlichkeit des Codes [66, 78].

3.1.2. Auswahl der Projekte

Zur Vorbereitung der Untersuchung wurden zunächst zahlreiche Open-Source-Projekte einer statischen Codeanalyse unterzogen. Für Python-Projekte wurde `SonarQube` [14] und für Apex-Projekte das Analysewerkzeug `PMD Apex` [7] verwendet. Erfasst wurden insbesondere die LOC sowie die COC auf Methodenebene. Ziel dieser Voranalyse war es, Projekte mit vergleichbarer Größe, jedoch unterschiedlicher COC zu identifizieren. Die Auswahl der final untersuchten Projekte erfolgte auf Basis der Softwaremetriken LOC und COC, um Größenunterschiede als Störfaktor zu minimieren und die Komplexität

gezielt variieren zu können. Zur Beantwortung der zweiten Forschungsfrage (RQ2) wurden zwei Programmiersprachen betrachtet. Python und Apex wurden ausgewählt, da für beide Sprachen geeignete Werkzeuge zur statischen Codeanalyse existieren und sie zugleich zwei unterschiedliche Pole darstellen: eine weitverbreitete Sprache mit hoher Trainingswahrscheinlichkeit für LLMs und eine weniger verbreitete Sprache.

3.1.3. Verwendete Large Language Modelle

Neben der Programmiersprache spielen auch die verwendeten LLMs eine zentrale Rolle. In dieser Arbeit wurden *Codestral-2501* und *Gemini 3 Pro* eingesetzt. Die Auswahl erfolgte unter anderem deshalb, da diese Modelle in bisherigen Arbeiten zu LLM-basierten Refactorings nur begrenzt untersucht wurden, wodurch eine Abgrenzung zu bestehenden Studien ermöglicht wird. Um die nicht deterministische Natur von LLMs [18] zu berücksichtigen, wurden alle Refactoring-Aufgaben zehnmal ausgeführt.

3.1.4. Bewertung der Refactorings

Ein Refactoring gilt als erfolgreich, wenn alle Tests bestehen, eine Änderung durchgeführt wurde und diese Änderung das Refactoring richtig durchgeführt hat. Fehlgeschlagen wäre das Refactoring, wenn keine Änderung durchgeführt oder mindestens ein Test nicht besteht oder das Refactoring nicht richtig implementiert wurde. Insgesamt ermöglicht dieses Forschungsdesign eine kontrollierte Untersuchung des Einflusses der COC, der Programmiersprache sowie des eingesetzten LLM auf die Qualität LLM-generierter Refactorings.

3.2. Auswahl und Beschreibung des Datensatzes

Die Auswahl des Datensatzes erfolgte anhand klar definierter Kriterien. Ziel war es, eine kontrollierte und vergleichbare Grundlage für das Forschungsexperiment zu schaffen. Die folgenden Anforderungen mussten erfüllt sein:

- Drei Projekte pro Programmiersprache (Python und Apex)
- Open-Source-Verfügbarkeit
- Vorhandene Testinfrastruktur
- Je ein Projekt mit niedriger, mittlerer und hoher COC
- Vergleichbare LOC innerhalb einer Sprache mit einer maximalen Differenz von 400 LOC zwischen dem kleinsten und dem größten Projekt

Die Kriterien wurden so gewählt, dass sowohl die Vergleichbarkeit als auch die Aussagekraft der Untersuchung gewährleistet sind. Die COC stellt die zentrale unabhängige Variable dar, während die LOC als Kontrollgröße dient, um Verzerrungen durch stark unterschiedliche Projektgrößen zu vermeiden. Die vorhandene Testinfrastruktur

ist notwendig, um die funktionale Korrektheit der Refactorings objektiv überprüfen zu können. Ohne die Testinfrastruktur wäre keine faire funktionale Validierung möglich gewesen. Obwohl die Auswahl kontrolliert anhand von LOC und COC erfolgte, kann die Generalisierbarkeit der Ergebnisse auf andere Softwareprojekte dadurch eingeschränkt sein. Für die Python-Projekte wurden zunächst potenzielle Open-Source-Projekte auf *GitHub* [42] unter Verwendung von Filtern bezüglich der Programmiersprache und Größe identifiziert und anschließend mithilfe statischer Codeanalyse hinsichtlich ihrer LOC und COC untersucht. Die dabei berücksichtigten Projekte sind in Tabelle 3.1 dargestellt. Aus dieser Menge entsprachen die Projekte *Colorama*, *pluggy* und *pathlib2* den definierten Auswahlkriterien am ehesten. Die Auswahl erfolgte dabei nicht nach der allgemeinen Bekanntheit der Projekte, sondern nach ihrer Eignung im Hinblick auf die festgelegten Vergleichskriterien. Bei den Apex-Projekten lief es ähnlich ab. Hervorgehoben haben sich aus den untersuchten Apex-Projekten die folgenden markierten in Tabelle 3.2: *apex-utilities*, *force-di*, *apex-dml-mocking*.

Tabelle 3.1.: Übersicht der analysierten Python-Projekte[1]

Projektname	COC	LOC	$\frac{COC}{LOC}$
netflix-proxy [13]	131	643	0.204
python-slugify [67]	64	250	0.256
tornado [87]	3908	27064	0.144
gunicorn [23]	1556	6208	0.251
pydantic [27]	6953	28908	0.241
sqlparse [4]	659	2857	0.231
pluggy [74]	231	1020	0.226
Jinja [71]	2090	8547	0.245
Colorama [86]	122	631	0.193
loguru [30]	924	3071	0.301
requests [73]	863	2961	0.291
sh [5]	649	2167	0.299
Markdown [75]	1690	4461	0.379
pyyaml [97]	1623	4328	0.375
tinydb [65]	159	768	0.207
Bottle [16]	974	2977	0.327
pathlib2 [49]	421	1000	0.421
yapf [45]	2366	5329	0.444
Colorama [86]	122	631	0.193
pluggy [74]	231	1020	0.226
pathlib2 [49]	421	1000	0.421

Tabelle 3.2.: Übersicht der analysierten Apex-Projekte[2]

Projektname	COC	LOC	$\frac{COC}{LOC}$
trigger-actions-framework [85]	48	2941	0.016
nebula-logger [50]	260	14256	0.018
at4dx-samplecode [8]	28	1518	0.018
apex-mdapi [21]	601	17791	0.034
fflib-apex-common-samplecode [9]	109	3101	0.035
automation-components [89]	3291	88863	0.037
dreamhouse-lwc [90]	25	623	0.040
apex-recipes [88]	393	8500	0.046
apex-utilities [72]	145	1940	0.075
httpcalloutframework [60]	82	586	0.140
sfdc-trigger-framework [54]	35	240	0.146
lightningflowcomponents [91]	36120	164221	0.220
fflib-apex-mocks [11]	1465	5094	0.288
force-di [12]	832	1976	0.421
fflib-apex-common [10]	3227	5995	0.538
apex-unified-logging [77]	117	211	0.555
apex-domainbuilder [76]	939	1624	0.578
apex-dml-mocking [82]	1413	2283	0.619
apex-lambda [17]	501	781	0.641
sendgrid-apex [80]	321	361	0.889
apex-rollup [47]	16914	12903	1.311
apex-utilities [72]	145	1940	0.075
force-di [12]	832	1976	0.421
apex-dml-mocking [82]	1413	2283	0.619

3.3. Auswahl und Definition der Refactorings

Die ausgewählten Refactorings sind in Tabelle 3.3 aufgeführt. Dabei war es im Vorhinein wichtig, jeweils zwei einfache, zwei mittlere und zwei schwere Refactoring-Aufgaben zu definieren. Die Einteilung der Schwierigkeitsstufen erfolgte anhand des Umfangs der erforderlichen Codeänderungen, des strukturellen Eingriffs in den Code sowie der Frage, ob die betroffenen Transformationsstellen explizit vorgegeben sind oder vom Modell selbst identifiziert werden müssen. Außerdem sollten die Refactoring-Aufgaben sowohl strukturelle als auch komplexitätsbezogene Veränderungen untersuchen. Zum einen sollte die Auswahl der Refactorings mit strukturellen Veränderungen in der Literatur [39] etabliert sein. Zum anderen musste eine objektive Bewertung der Ergebnisse möglich sein. Durch die systematische Auswahl unterschiedlicher Schwierigkeitsgrade wird sichergestellt, dass die Leistungsfähigkeit der Modelle nicht nur bei trivialen Umbenennungen, sondern auch bei komplexeren Designänderungen untersucht wird.

Tabelle 3.3.: Ausgewählte Refactoring-Aufgaben

Aufgabe	**Beschreibung**	**Stufe**
Methode umbenennen	Umbenennen einer Methode im gesamten Projekt	einfach
Variable inline setzen	Entfernen einer Variablen, indem ihr Wert direkt in den Ausdruck eingesetzt wird	einfach
Getter und Setter erstellen	Aufrufe einer Funktion zu Getter und Setter umwandeln	mittel
Guard Clauses erstellen	Umwandeln verschachtelter `if`-Anweisungen in Guard Clauses	mittel
Strategy Pattern anwenden	Auslagern von Hilfslogik durch Anwendung des Strategy Patterns	schwierig
Cognitive Complexity reduzieren	COC Reduktion zur besseren Lesbarkeit	schwierig

Die einfachen Refactorings umfassen strukturelle Änderungen mit klar definiertem Umfang. Dazu gehören das Umbenennen einer Methode im gesamten Projekt sowie das Inline-Setzen einer Variable. Beide Aufgaben erfordern primär syntaktische Konsistenz und eine korrekte Anpassung aller Referenzen, ohne tiefgreifende strukturelle Änderungen vorzunehmen. Die mittleren Refactorings betreffen bereits komplexere Umstrukturierungen. Hierzu zählen das Erstellen von Getter- und Setter-Methoden sowie das Umwandeln verschachtelter `if`-Anweisungen in *Guard Clauses*. Wobei nicht explizit angegeben wird, welcher Codeabschnitt diese Aufgaben betrifft. So muss das LLM selbst entscheiden, wo dieses Refactoring angewendet werden soll. Diese Aufgaben greifen stärker in die Struktur des Codes ein und beeinflussen die Lesbarkeit. Somit wird die Praxisnähe des Experiments erweitert, da das LLM auch die Identifikation der geeigneten Transformationsstellen eigenständig vornehmen muss. Die schwierigen Refactorings umfassen strukturelle Veränderungen auf Design-Ebene. Dazu gehört die Anwendung

des *Strategy Pattern*s zur Auslagerung von Logik sowie die Reduktion der COC. Bei der Reduktion der COC wird auch nicht, so wie bei dem Guard Clauses Refactoring, explizit angegeben, welche genauen Codeabschnitte refactored werden sollen. Alle Refactorings wurden unter der Voraussetzung durchgeführt, dass das externe Verhalten des Systems unverändert bleibt. Ein Refactoring wurde nur dann als erfolgreich gewertet, wenn sowohl die strukturelle Änderung korrekt umgesetzt als auch alle Tests bestanden wurden. Unter welche Kriterien die strukturellen Änderungen bewertet werden, wird in Abschnitt 3.6 eingegangen.

3.4. Auswahl der Large Language Modelle

Neben den Refactorings stellen die verwendeten LLMs eine zentrale unabhängige Variable des Forschungsexperiments dar. Die Auswahl der Modelle erfolgte anhand mehrerer Kriterien. Zum einen sollten die Modelle aktuell und leistungsfähig im Bereich Codegenerierung sein. Zum anderen sollten gewisse Unterschiede zwischen den Modellen bestehen. In dieser Arbeit wurden *Codestral-2501* und *Gemini 3 Pro* eingesetzt. *Codestral* basiert auf einer Modellfamilie, die speziell für Codegenerierungsaufgaben optimiert wurde [25]. Modelle mit Code-spezifischem Training zeigen in Benchmarks wie *HumanEval* eine verbesserte Fähigkeit zur syntaktisch korrekten und funktionalen Codeerzeugung [22]. Es ist jedoch wenig bekannt, wie solche LLMs sich in realen Codebasen zurechtfinden und wie gut sie Refactoring-Aufgaben richtig umsetzen können. *Gemini 3 Pro* wurde ausgewählt, da es als leistungsfähiges multimodales Modell gilt und in praktischen Anwendungen im Bereich Softwareentwicklung zunehmend eingesetzt wird [1].

3.5. Prompt Engineering

Da LLMs stark von der Formulierung der Eingabeaufforderung abhängen, stellt das Prompt-Engineering einen zentralen Bestandteil des Forschungsexperiments dar. Ziel war es, eine stabile und reproduzierbare Prompt-Struktur zu entwickeln, die konsistent auf alle untersuchten Modelle angewendet werden kann. Um einen Bias im späteren Experiment zu vermeiden, wurde das Prompt-Engineering ausschließlich an Testprojekten und mit einem separaten LLM durchgeführt. Für die Durchführung der Refactoring-Aufgaben wurde ein *Few-Shot-Prompting*-Ansatz gewählt. *Few-Shot-Prompting* beschreibt die Bereitstellung einer begrenzten Anzahl exemplarischer Beispiele innerhalb des Prompts [79]. Dieser Ansatz war im vorliegenden Experiment besonders relevant, da konkrete Beispiele zu den jeweiligen Refactorings bereitgestellt wurden, um das Modellverhalten gezielt zu steuern. Darüber hinaus wurde entschieden, den Prompt auf Englisch zu verfassen, da LLMs Aufgaben in dieser Sprache nachweislich am zuverlässigsten bearbeiten [32, 31].

3.5.1. Testsetup und Zuverlässigkeitskriterium

Der Prompt-Engineering-Prozess orientierte sich an Schulhoff u. a. [79], wonach ein Prompt so lange getestet und angepasst wird, bis er bei einem Testprojekt zuverlässig funktioniert. Dieses Vorgehen wurde übernommen und ist anschaulich in Abbildung 3.1 dargestellt. Als Testprojekte wurden *tinydb* (Python) und *apex-domainbuilder* (Apex) verwendet, da beide Projekte lauffähige Testinfrastruktur besitzen. Als Kriterium, dass ein Prompt eine Aufgabe zuverlässig erledigen kann, wurde festgelegt, dass ein Prompt bei einem Refactoring im jeweiligen Testprojekt in zehn Ausführungen mindestens 8/10 erfolgreiche Durchläufe erreicht. Außerdem wurde die Methode verfeinert. So wurde unter anderem *Meta-Prompting* eingesetzt, bei dem auf Basis des definierten Ziels ein verbesserter Prompt mithilfe eines LLM generiert wurde [79]. Der initiale Meta-Prompt ist in Listing 3.1 zu erkennen. Aus diesem Meta-Prompt entstand daraufhin der erste Entwurf in Listing 3.2.

Listing 3.1.: Meta-Prompt zur Generierung eines Rename-Prompts (Initialversion)

```
1  You are a prompt engineer. Generate a single refactoring prompt (no explanation, no
   ↪  commentary) for the refactoring task "Rename" in a Python project.
2
3  The generated prompt must follow this exact structure and content order:
4
5  1) Title line: "Rename"
6  2) Line: "Task Description"
7  3) Add an introductory paragraph that defines the assistant's role and constraints
   ↪  (refactoring task, strict adherence to the given rules, no extra changes).
   ↪  Briefly summarize the refactoring objective and scope (project-wide application
   ↪  where needed) for the chosen refactoring type in the target programming
   ↪  language.
8  4) Section heading: "Please note:" followed by exactly these rules (verbatim, same
   ↪  wording and punctuation):
9     1. The code must still function perfectly after the refactoring.
10    2. All imports, references, and function calls must be updated accordingly.
11    3. The style, formatting, and structure of the code must be preserved.
```

Listing 3.2.: Aus dem Meta-Prompt generierter Prompt

```
Rename
Task Description
You are a strict code-refactoring engine. You follow all rules exactly as written. Your task is to perform a refactoring change in a Python project. A function needs to be renamed.
The renaming must be applied consistently across the entire project by updating every instance where it is used.

Please note:
1. The code must still function perfectly after the renaming.
2. All imports, references, and function calls must be updated accordingly.
3. The style, formatting, and structure of the code must be preserved.
```

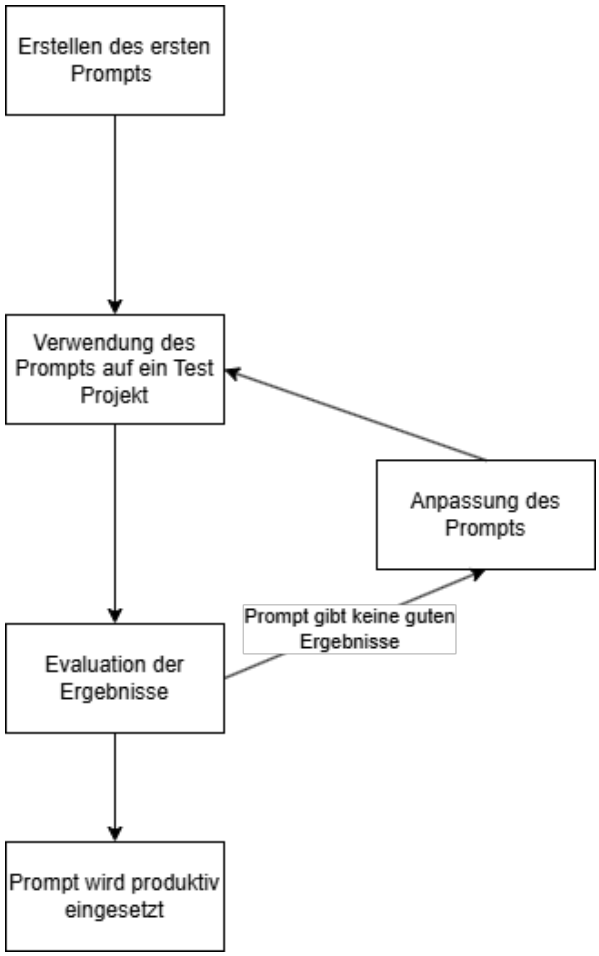

Abbildung 3.1.: Prompt Engineering Überblick

Listing 3.3.: Rollenbeschreibung für das Rename-Refactoring

```
1 You are a strict code refactoring engine. You follow all rules exactly as written.
2 Your task is to perform a refactoring change in an Apex project.
3 A method needs to be renamed.
4 The renaming must be applied consistently across the entire project by updating
5 every instance where it is used.
```

3.5.2. Prompt-Struktur und Refinement

Aus folgenden Bestandteilen setzte sich der entwickelte Prompt zusammen: einer klar definierten Rollenbeschreibung (vgl. Listing 3.3) für das LLM, strukturellen Anforderungen an das Refactoring (vgl. Listing 3.5), zwei konkrete Beispiele zur Veranschaulichung (vgl. Listing 3.4), der eigentlichen Aufgabenbeschreibung sowie den gesamten Code des Projekts. Ein vollständiger Prompt ohne den Code ist in Abschnitt A.1. Um systematisch den Prompt iterativ zu verbessern, wurden beobachtete Fehler gesammelt und in konkrete Regeln überführt. Ein Ansatz, der auf Skreta u. a. [83] zurückgeht. Dieses Tuning erfolgte mit *Gemini 2.5 Flash Lite* um prompt-varianten schnell und kostengünstig iterieren zu können. Typische Fehlermuster umfassten etwa unerwartete Änderungen außerhalb des Refactoring-Ziels, syntaktisch oder semantisch fehlerhaften Code, der die Tests nicht bestand, oder Ausgaben ohne tatsächliche inhaltliche Anpassungen. Auch unlesbare Modellantworten, etwa durch erklärende Fülltexte anstelle von reinem Code, wurden in diesem Prozess erfasst. Aus diesen Beobachtungen entstanden die Regeln in Listing 3.5. Zudem kam die Technik der *Prompt-Repetition* zum Einsatz: Dabei wird die Aufgabenstellung innerhalb des Prompts explizit wiederholt, was nachweislich dazu beiträgt, dass das LLM sie besser berücksichtigt [57].

3.6. Validierung

Dieser Abschnitt beschreibt die Testinfrastruktur und das Vorgehen zur Validierung der Refactorings. Dabei wird erläutert, wie die Korrektheit der erzeugten Refactorings überprüft wird und welche Testinfrastruktur für diese Überprüfung geschaffen wurde.

3.6.1. Testinfrastruktur

Um faire Vergleiche zwischen den Projekten zu ermöglichen, wurden fehlende Tests mit *GitHub Copilot* [43] generiert, so dass jedes Projekt pro Sprache eine ähnliche Testanzahl hat. Als Zielmarke diente jeweils das Python-Projekt mit den meisten Tests (426) bzw. das Apex-Projekt mit den meisten Tests (129). Tabelle 3.4 zeigt die Aufteilung: Während *colorama* fast ausschließlich generierte Tests (377/420) nutzt, bringt *pathlib2* viele eigene Tests mit (322/426). Apex-Projekte hatten bereits solide Testbestände, weshalb nur wenige ergänzt werden mussten. Die Testabdeckung ist bei Python sehr hoch (>99%), bei Apex sehr gut vergleichbar (84–88%). Damit liegt eine robuste Grundla-

Listing 3.4.: Beispiel im Few-Shot-Prompt

````
Example 1:
Original Code:
File `calculate.py`:
```python
def calculate_sum(a, b):
    return a + b

print(calculate_sum(2, 3))
```

Task:
1. Rename the method `calculate_sum` in the file: `calculate.py` to `add`.
2. Update every relevant occurrence across the entire project.
3. Output format (for EACH modified file):

File `calculate.py`:
```python
def add(a, b):
    return a + b

print(add(2, 3))
```
````

Listing 3.5.: Regeln welche in jedem Prompt stehen

```
Please note:
1. The code must still function perfectly after the refactoring.
2. All imports, references, and function calls must be updated accordingly.
3. The style, formatting, and structure of the code must be preserved.
4. Do not introduce any new functionality, logic changes, or unrelated formatting
↪  changes.
5. For every modified file, return the complete updated file content to ensure that
↪  no changes are overlooked.
6. The semantics and behavior of the code must not change.
7. Only include files that required changes.
8. Respond with the code blocks only. Your entire response must be
↪  machine-readable. Do not include any conversational filler.
9. If you are unsure, do not guess. Only modify code when the change is certain.
10. Every response must include meaningful modifications, improvements, or
↪  transformations. Returning the same code, even partially or with superficial
↪  edits, is strictly prohibited.
11. Your final output must pass the existing test suite (pytest). If your changes
↪  would cause any test to fail, revise your refactor until all tests pass.
```

ge für die funktionale Validierung der Refactorings vor. Eine höhere Testabdeckung in den Apex-Projekten konnte nicht erreicht werden, da *GitHub Copilot* [43] ausschließlich fehlerhafte Tests generierte, die weder deploybar noch stabil lauffähig waren.

Tabelle 3.4.: Testaufteilung zwischen vorhandenen und generierten Tests pro Projekt

Projekt	Vorhandene Tests	Generierte Tests	Gesamt	Testabdeckung
Python-Projekte				
colorama	43	377	420	99%
pluggy	126	300	426	100%
pathlib2	322	104	426	99%
Apex-Projekte				
apex-utilities	129	0	129	88%
force-di	61	66	117	84%
apex-dml-mocking	92	37	129	87%

3.6.2. Validierung der Refactorings

Die Validierungsstrategie variierte je nach Refactoring-Typ, wie Tabelle 3.5 zeigt. Alle Refactorings wurden mittels Regressionstests validiert, indem die Tests vor und nach dem Refactoring ausgeführt wurden. Für die Refactorings Rename und Getter-Setter (nur bei Apex) mussten die Tests jedoch zunächst angepasst werden, da diese Refactorings die öffentliche Schnittstelle des Codes verändern. Durch das Umbenennen einer Methode oder das Ersetzen eines direkten Attributzugriffs durch Getter- und Setter-Methoden würden die unveränderten Tests fehlschlagen, obwohl das Refactoring korrekt durchgeführt wurde. Bei Python war dies für das Getter-Setter-Refactoring nicht notwendig, da durch die Verwendung von `@property`-Dekoratoren die Schnittstelle unverändert bleibt. Der Zugriff auf Attribute erfolgt syntaktisch weiterhin über `obj.attribute`, auch wenn intern Getter- und Setter-Methoden aufgerufen werden.

Tabelle 3.5.: Validierungsstrategie je Refactoring-Typ

Refactoring	**Validierungsmethode**
Rename	Regressionstests + Angepasste Tests
Inline Variable	Regressionstests + manuelle Prüfung
Getter-Setter	Regressionstests + Angepasste Tests (nur Apex), Regressionstests + manuelle Prüfung (nur Python)
Guard Clauses	Regressionstests + manuelle Prüfung
Strategy Pattern	Regressionstests + manuelle Prüfung
COC-Reduktion	Regressionstests + COC-Messung vor/nach

Für die manuelle Prüfung wurden pro Refactoring-Aufgabe Kriterien definiert. Erst

wenn das Refactoring alle Kriterien erfüllte, wurde es als korrekt bewertet. So galten für die Refactorings: Inline Variable, Getter-Setter, *Guard Clauses* und *Strategy Pattern* folgende Kriterien.

Inline Variable

- Die ursprüngliche Variable wurde vollständig entfernt.
- Der zuvor gespeicherte Ausdruck wurde direkt an den entsprechenden Stellen im Code eingesetzt.
- Alle Referenzen der Variable wurden korrekt ersetzt.

Getter-Setter-Refactoring (Python mit @property)

- Für die betroffene Instanzvariable wurde eine Getter-Methode erstellt.
- Für die betroffene Instanzvariable wurde eine Setter-Methode erstellt.

Guard Clauses Eine geänderte Datei gilt als verbessert, wenn:

a) die Anzahl der Guard Clauses (frühe `return`, `continue`, `break`, `throw`-Statements) sich erhöht hat, **oder**

b) die maximale `if`-Verschachtelungstiefe reduziert wurde oder gleich geblieben ist

Strategy Pattern

- Die ursprüngliche Entscheidungslogik wurde in separate Strategien ausgelagert.
- Es existiert eine gemeinsame Schnittstelle oder Basisklasse für die Strategien.
- Die konkrete Strategie wird zur Laufzeit ausgewählt und verwendet.

Um die COC-Reduktion zu validieren wurde die COC vor dem Refactoring und nach dem Refactoring gemessen und daraufhin verglichen. Das Refactoring galt als korrekt, wenn die COC nach dem Refactoring geringer war als davor. Der Aufwand aller dieser Prüfungen fällt pro Ausführung an und variiert je nach Refactoring-Typ und Änderungsumfang. Dadurch war die Anzahl der untersuchten Ausführungen praktisch durch den verfügbaren Zeitrahmen begrenzt.

3.7. Ablauf des Experiments

Nachdem die Refactoring-Aufgaben, die Projekte und die LLMs ausgewählt wurden und die Prompts feststanden, waren fast alle Vorbereitungen getroffen um mit dem Experiment beginnen zu können. Um die Vergleichbarkeit zwischen den Projekten weiter zu erhöhen, wurden zunächst alle Kommentare aus den Quellcode-Dateien entfernt, die an das LLM übergeben wurden. Dadurch sollte sichergestellt werden, dass zusätzliche Informationen in Form von Kommentaren keinen Einfluss auf die Generierung der Refactorings haben. Der experimentelle Ablauf wurde für alle Projekte und Refactoring-Aufgaben einheitlich durchgeführt, um die Vergleichbarkeit zwischen den Modellen und Programmiersprachen sicherzustellen. Der schematische Ablauf des Experiments ist in Abbildung 3.2 zu erkennen. Zu Beginn wurde der ursprüngliche Projektzustand gesichert. Anschließend wurde die COC des relevanten Codes mittels statischer Analyse erfasst. Diese Messung diente als Ausgangswert für den späteren Vergleich. Daraufhin wurde die jeweilige Refactoring-Aufgabe zusammen mit dem definierten Few-Shot-Prompt an das entsprechende LLM übermittelt. Das Modell generierte daraufhin eine modifizierte Version des Codes, in der das geforderte Refactoring umgesetzt werden sollte. Der generierte Code wurde anschließend in das Projekt integriert. Hier entstand ein kleiner Unterschied zwischen den Apex- und Python-Projekten. In den Python-Projekten wurde lediglich die syntaktische Korrektheit mithilfe von Tests validiert, während in Apex der Code zunächst deployed werden musste und daraufhin die Tests erfolgten. Bei Python-Projekten wurden hierfür `pytest`-Tests ausgeführt, bei Apex-Projekten die entsprechenden Klassentests. Danach wurde die Umsetzung des Refactorings manuell validiert. Pro Refactoring wurde dieses Vorgehen zehnmal wiederholt. Jedes Ergebnis jeder Iteration wurde daraufhin dokumentiert.

3.8. Technische Umsetzung

Für die technische Umsetzung wurde pro Programmiersprache je ein Python-Skript entwickelt. Beide Skripte folgen demselben grundlegenden Ablauf, unterscheiden sich jedoch in den sprachspezifischen Details der Testausführung und Validierung. Die in Abbildung 3.2 dargestellten Schritte – Prompt senden, Code generieren, in das Projekt integrieren und Tests ausführen – wurden vollständig automatisiert.

3.8.1. Projektstruktur

Der Zugriff auf die LLMs erfolgte über *API-Keys* der jeweiligen Anbieter, die über Umgebungsvariablen geladen wurden. Abhängig vom gesetzten *API-Key* wurden automatisch das entsprechende Modell sowie die zugehörige Client-Bibliothek ausgewählt und initialisiert. Für *Gemini 3 Pro* wurde `google-genai`, für *Codestral-2501* `mistralai` verwendet. Weitere genutzte Bibliotheken waren `subprocess`, `re`, `difflib` sowie `pathlib`.

3.8.2. Automatisierter Ablauf

Zu Beginn jeder Refactoring-Aufgabe wird zunächst ein Backup des Projekts angelegt. Das ist wichtig, damit das Projekt nach jeder Iteration in seinen Ausgangszustand zurückversetzt werden kann. Anschließend liest das Skript den Prompt aus einer zugehörigen `.txt`-Datei ein und ergänzt ihn um die Projektstruktur sowie den vollständigen Quellcode. Dieser finale Prompt wird dann an das LLM gesendet. Die Antwort des Modells wird mithilfe eines regulären Ausdrucks geparst. Das LLM gibt den Code im Format `File <Dateiname>: python ...` zurück, woraus Dateiname und Codeinhalt extrahiert werden:

Listing 3.6.: Regex zum Parsen der LLM-Antwort

```
pattern = r"File\s+([^`]+)`:\s*```python\s*(.*?)\s*```"
matches = re.findall(pattern, response_text, re.DOTALL)
```

Liefert der Regex keine Treffer, gilt die Iteration als fehlgeschlagen und wird übersprungen. Andernfalls wird die jeweilige Datei vollständig mit dem generierten Inhalt überschrieben. Testdateien werden dabei explizit ausgenommen – es wird ausschließlich Produktionscode verändert. Um zu prüfen, ob das LLM überhaupt eine Änderung vorgenommen hat, wird nach jeder Iteration ein Diff zwischen dem Backup und dem refactored Code berechnet. Dabei werden Leerzeilen und Whitespace normalisiert, um rein formatierungsbedingte Unterschiede herauszufiltern. `pytest` wird anschließend über `subprocess` aufgerufen, und das Ergebnis wird anhand des Exit-Codes ausgewertet. Die Testabdeckung wurde mit `pytest-cov` gemessen und musste bei Python-Projekten 99 oder 100 Prozent betragen, um eine vollständige Validierung sicherzustellen. Pro Iteration werden folgende Ergebnisse in einer Zusammenfassungsdatei festgehalten: ob die Tests bestanden wurden, ob eine Änderung am Code vorgenommen wurde und ob das Refactoring korrekt implementiert wurde. Zusätzlich wird der Token-Verbrauch dokumentiert. Eine beispielhafte Ausgabe sieht wie folgt aus: Die Ergebnisse wurden an-

Listing 3.7.: Beispielhafte Zusammenfassungsdatei

```
iteration 1 passed test passed diff passed ref passed Tokens: ...
iteration 2 failed test passed diff failed ref failed Tokens: ...
```

schließend manuell überprüft und in eine CSV-Datei übertragen und bilden die Grundlage für die Auswertung in Kapitel 4. Jedes Refactoring wurde pro Modell und Projekt zehnmal ausgeführt, um die nicht deterministische Natur der LLMs zu berücksichtigen und zufällige Schwankungen auszugleichen.

3.8.3. Umsetzung in Apex

Die Umsetzung für Apex folgt prinzipiell demselben Ablauf, weist jedoch einige Besonderheiten auf. Der Regex-Parser erwartet hier das Format `File <Dateiname>: apex ...` , da Apex-Dateien die Endung `.cls` oder `.trigger` tragen. Ein zentraler Unterschied liegt in der Testausführung. Apex-Code kann nicht lokal ausgeführt werden, sondern muss zunächst in eine Salesforce-Sandbox deployed werden. Das Deployen sowie das anschließende Ausführen der Klassentests wurden ebenfalls über `subprocess` automatisiert: Ein fehlgeschlagener Deploy wurde direkt als Testfehler gewertet, ohne

Listing 3.8.: Automatisierter Deploy und Testlauf in Apex

```
subprocess.run(
    "sf project deploy start --ignore-conflicts",
    capture_output=True, shell=True, env=env
)
subprocess.run(
    "sf apex run test --wait 10 --result-format human",
    capture_output=True, shell=True, env=env
)
```

dass die Klassentests überhaupt ausgeführt wurden. Für die Messung der COC wurde das statische Analysewerkzeug `PMD` eingesetzt. Die XML-Ausgabe von `PMD` wurde mittels `xml.etree.ElementTree` geparst, um die Komplexitätswerte vor und nach dem Refactoring zu extrahieren und zu vergleichen.

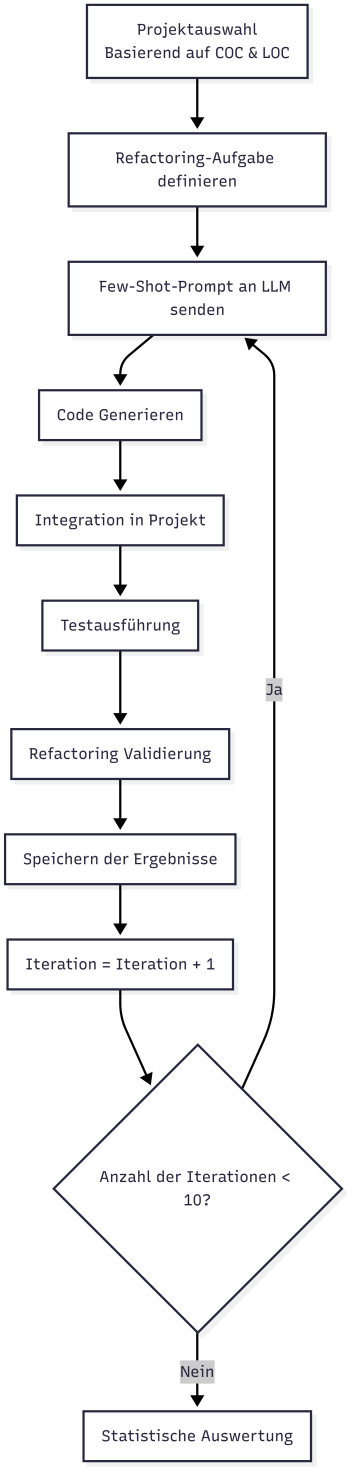

Abbildung 3.2.: Ablauf des Experiments Überblick

4. Ergebnisse

In diesem Kapitel werden alle Ergebnisse des durchgeführten Experiments dargestellt. Zunächst wird ein Überblick über das Gesamtexperiment gegeben, bevor die Erfolgsraten der Refactorings nach Modell, Komplexitätsstufe, Programmiersprache und Refactoring-Typ ausgewertet werden. Anschließend folgt eine detaillierte Analyse der erzeugten Codeänderungen, der korrekten Umsetzung der Refactorings sowie der funktionalen Korrektheit auf Basis der Testergebnisse. Darüber hinaus werden typische Fehlermuster und beispielhafte Fehlrefactorings untersucht. Abschließend werden die Stabilität der Ergebnisse sowie die zentralen Befunde des Experiments zusammengefasst.

4.1. Überblick über das Gesamtexperiment

Im Rahmen des Experiments wurden insgesamt sechs Projekte untersucht, jeweils drei in Python und drei in Apex. Für jedes Projekt wurden sechs unterschiedliche Refactoring-Aufgaben definiert, die sich in einfache, mittlere und schwere Refactorings unterteilen lassen (vgl. Tabelle 3.3). Jede Refactoring-Aufgabe wurde pro Modell zehnmal ausgeführt, um die nicht deterministische Natur der LLMs zu berücksichtigen. In dieser Arbeit bezeichnet eine Iteration eine einzelne Refactoring-Ausführung. Eine Iteration umfasst somit genau einen Versuch eines Modells, ein bestimmtes Refactoring in einem Projekt umzusetzen. Insgesamt ergibt sich damit folgende Anzahl an Experimentdurchläufen:

- 6 Projekte
- 6 Refactorings pro Projekt
- 2 Modelle
- 10 Iterationen

Dies entspricht insgesamt 720 einzelnen Refactoring-Ausführungen. Ein Refactoring wurde unter folgenden Bedingungen als erfolgreich gewertet:

- Das geforderte Refactoring wurde im Rahmen der manuellen Validierung als korrekt umgesetzt bewertet.
- alle bestehenden Tests wurden erfolgreich bestanden

Im Folgenden werden die Ergebnisse zunächst hinsichtlich der funktionalen Korrektheit analysiert. Anschließend erfolgt eine differenzierte Betrachtung nach Modell, Komplexitätsstufe, Refactoring-Typ sowie Programmiersprache.

4.2. Erfolgsrate aus dem Experiment

In diesem Abschnitt wird die Erfolgsrate in Abhängigkeit verschiedener Attribute (z. B. Modell, Programmiersprache, Projekt und Refactoring-Typ) analysiert. Die Erfolgsraten beziehen sich auf die durchschnittliche Erfolgsrate der Refactoring-Ausführungen. Sie ergibt sich aus dem Anteil erfolgreicher Refactoring-Ausführungen an allen durchgeführten Ausführungen innerhalb der jeweiligen Kategorie. Als Erfolg gilt eine Iteration, wenn das Refactoring richtig umgesetzt wurde und alle Tests bestehen. Die Erfolgsrate entspricht damit dem Anteil erfolgreicher Iterationen an allen durchgeführten Iterationen. Methodisch ist dieses Maß vergleichbar mit `pass@1`, da pro Iteration genau ein Refactoring-Vorschlag bewertet wird [22].

4.2.1. Erfolgsrate nach Modell

Wird die Gesamterfolgsrate beider Modelle über alle Projekte, Refactoring-Typen und Sprachen hinweg betrachtet, zeigt sich ein deutlicher Unterschied. *Gemini 3 Pro* erzielte eine durchschnittliche Erfolgsrate von 94,72%, während *Codestral-2501* (51,94%) deutlich darunter lag (vgl. Abbildung 4.1). Der Unterschied beträgt damit über 44 Prozentpunkte und zieht sich konsistent durch nahezu alle untersuchten Kombinationen. Lediglich bei einfachen Refactoring-Typen wie *Rename* und *Getter-Setter* konnte *Codestral* ähnliche Werte erzielen. Bei schwierigen Refactorings wie der *COC-Reduktion* und dem *Strategy Pattern* hingegen brach die Erfolgsrate von *Codestral* deutlich ein, während *Gemini 3 Pro* auch dort stabile Ergebnisse lieferte (vgl. Tabelle 4.3).

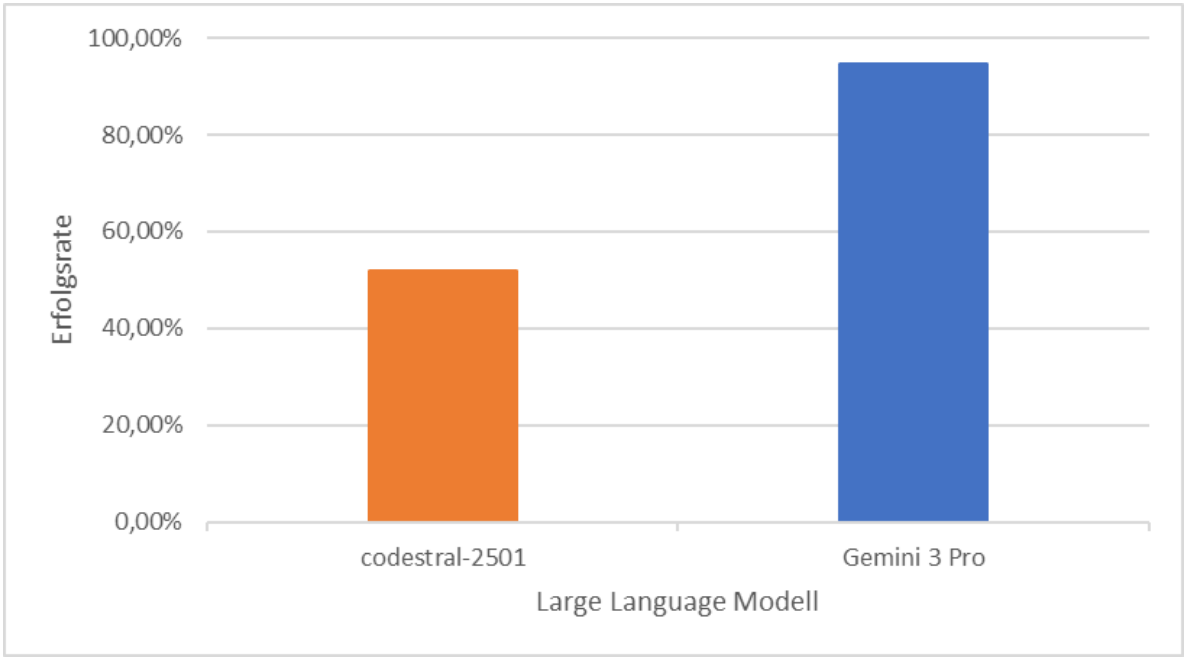

Abbildung 4.1.: Erfolgsrate nach Modell

4.2.2. Erfolgsrate nach Komplexitätsstufe

Um zu untersuchen, ob die Komplexität eines Projekts einen Einfluss auf die Erfolgsrate der Modelle hat, wurden die Ergebnisse nach den drei Komplexitätsstufen gering, mittel und komplex aufgeschlüsselt. Tabelle 4.1 zeigt die durchschnittliche Erfolgsrate beider Modelle und Sprachen je Komplexitätsstufe. Dieser Abschnitt ist vor allem für die Beantwortung von RQ 1 relevant. Auf den ersten Blick zeigt sich kein eindeutiger Zusammenhang zwischen der Projektkomplexität und der Erfolgsrate. *Gemini 3 Pro* erzielt über alle drei Stufen hinweg konstant hohe Werte und fällt lediglich bei komplexen Projekten leicht auf 90,0% ab. *Codestral-2501* hingegen zeigt ein gegensätzliches, auf den ersten Blick unerwartetes Muster: Die Erfolgsrate steigt mit zunehmender Komplexität leicht an, während sie bei *Gemini 3 Pro* geringfügig abnahm. Insgesamt lässt sich anhand der deskriptiven Ergebnisse kein eindeutiger Zusammenhang zwischen der Projektkomplexität und der Erfolgsrate erkennen. Die beobachteten Unterschiede zwischen den Modellen deuten jedoch darauf hin, dass der Einfluss der Komplexität modellabhängig sein könnte. Eine statistische Überprüfung dieser Zusammenhänge erfolgt in Abschnitt 4.3.

Tabelle 4.1.: Durchschnittliche Erfolgsrate nach Komplexitätsstufe, Programmiersprache und Modell

	Codestral-2501			**Gemini 3 Pro**			**Gesamt**
Komplexitätsstufe	**Apex**	**Python**	**Gesamt**	**Apex**	**Python**	**Gesamt**	
Gering	38,33% (n=60)	50,00% (n=60)	44,17% (n=120)	98,33% (n=60)	96,67% (n=60)	97,50% (n=120)	70,83% (n=240)
Mittel	51,67% (n=60)	51,67% (n=60)	51,67% (n=120)	100,00% (n=60)	93,33% (n=60)	96,67% (n=120)	74,17% (n=240)
Komplex	50,00% (n=60)	70,00% (n=60)	60,00% (n=120)	80,00% (n=60)	100,00% (n=60)	90,00% (n=120)	75,00% (n=240)
Gesamtergebnis	46,67% (n=180)	57,22% (n=180)	51,94% (n=360)	92,78% (n=180)	96,67% (n=180)	94,72% (n=360)	73,33% (n=720)

n = Anzahl der einzelnen Refactoring-Ausführungen pro Zelle

4.2.3. Erfolgsrate nach Programmiersprache

Um zu untersuchen, ob die Programmiersprache einen Einfluss auf die Leistung der Modelle hat, wurden die Ergebnisse nach Python und Apex aufgeschlüsselt. Dies ist vorwiegend für RQ2 relevant. Abbildung 4.2 zeigt die durchschnittliche Erfolgsrate beider Modelle je Programmiersprache über alle Projekte und Refactoring-Typen hinweg. *Gemini 3 Pro* erzielt in beiden Sprachen eine konstant hohe Erfolgsrate von 92,8 % in Apex und 96,7 % in Python. Der Unterschied ist gering und deutet darauf hin, dass das Modell unabhängig von der Programmiersprache zuverlässig arbeitet. *Codestral-2501* zeigt hingegen deutlichere Unterschiede: Mit 46,7 % in Apex fällt die Erfolgsrate merklich niedriger aus als in Python, wo 57,2 % erreicht werden. Der Unterschied von 10,5% legt nahe, dass *Codestral-2501* mit der Syntax und Semantik von Python bes-

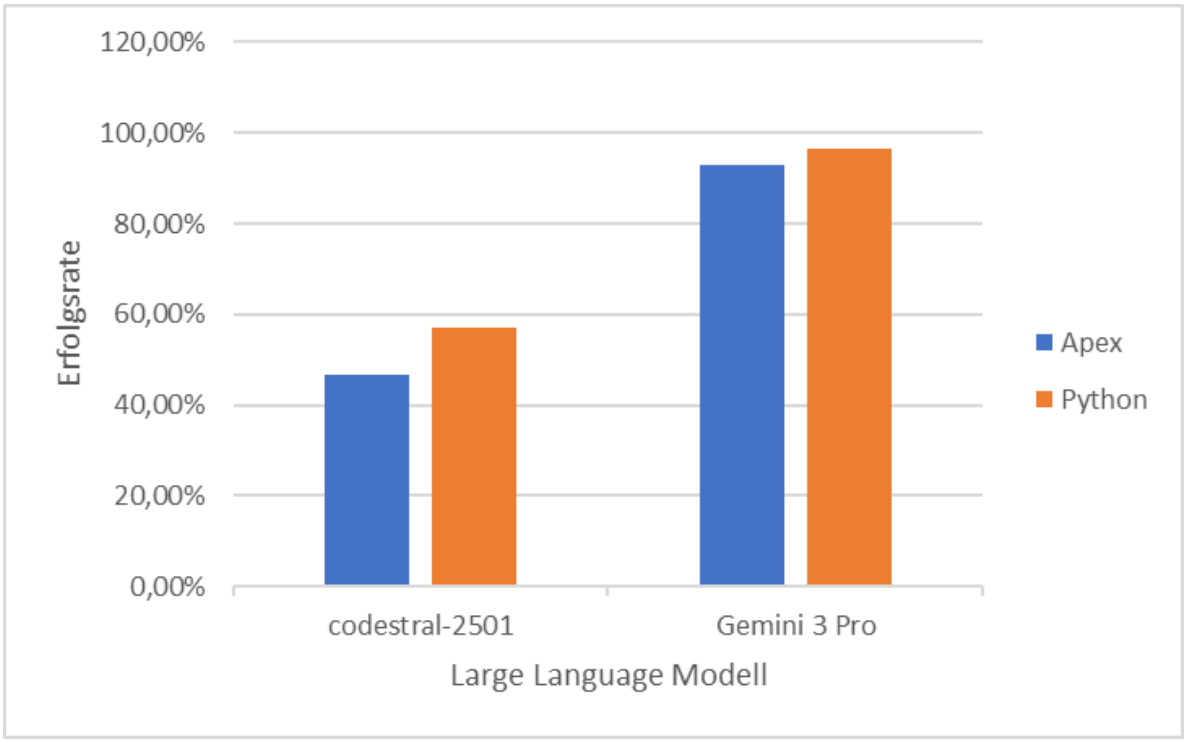

Abbildung 4.2.: Erfolgsrate nach Sprache

ser zurechtkommt als mit Apex. Tabelle 4.2 zeigt die Aufschlüsselung nach Refactoring und Programmiersprache. Dabei wird deutlich, dass der Unterschied zwischen den Sprachen vor allem bei schwierigen Refactorings zum Tragen kommt. Bei *Getter-Setter* erreicht *Codestral* in Python 100 %, während in Apex nur 33,3 % gelingen. Auch bei *Guard Clauses* zeigt sich ein ähnliches Muster: Python liegt mit 50 % deutlich über Apex mit 26,7 %. Umgekehrt schneidet *Codestral* bei *Rename* in Apex besser ab als in Python (93,3 % vs. 76,7 %). Dieser Befund deutet darauf hin, dass die Leistung stark vom Zusammenspiel zwischen Refactoring-Typ und Programmiersprache abhängt und nicht allein durch die Sprache erklärt werden kann. Für *Gemini 3 Pro* sind die Unterschiede zwischen den Sprachen deutlich geringer. Lediglich bei *Guard Clauses* (Apex: 80 %, Python: 100 %) und *COC-Reduktion* (Apex: 76,7 %, Python: 86,7 %) zeigen sich leichte Unterschiede zugunsten von Python. Das deutet darauf hin, dass *Gemini 3 Pro* sprachunabhängiger arbeitet als *Codestral-2501*. Insgesamt lässt sich festhalten, dass die Programmiersprache zwar einen messbaren Einfluss auf die Erfolgsrate hat, dieser jedoch stark vom Modell und vom Refactoring-Typ abhängt. *Gemini 3 Pro* zeigt in beiden Sprachen eine robuste Leistung, während *Codestral-2501* in Python tendenziell bessere Ergebnisse erzielt, jedoch auch dort bei schwierigen Refactorings deutlich hinter *Gemini 3 Pro* zurückfällt.

Tabelle 4.2.: Erfolgsrate nach Programmiersprache, Refactoring-Typ und Modell

Refactoring	**Codestral-2501**		**Gemini 3 Pro**	
	Apex	Python	Apex	Python
Rename	93,3 % (n=30)	76,7 % (n=30)	100,0 % (n=30)	100,0 % (n=30)
Inline Variable	76,7 % (n=30)	86,7 % (n=30)	100,0 % (n=30)	93,3 % (n=30)
Getter-Setter	33,3 % (n=30)	100,0 % (n=30)	100,0 % (n=30)	100,0 % (n=30)
Guard Clauses	26,7 % (n=30)	50,0 % (n=30)	80,0 % (n=30)	100,0 % (n=30)
Strategy Pattern	36,7 % (n=30)	20,0 % (n=30)	100,0 % (n=30)	100,0 % (n=30)
COC-Reduktion	13,3 % (n=30)	10,0 % (n=30)	76,7 % (n=30)	86,7 % (n=30)
Gesamt	46,7 % (n=180)	57,2 % (n=180)	92,8 % (n=180)	96,7 % (n=180)

n = Anzahl der einzelnen Refactoring-Ausführungen pro Zelle

4.2.4. Erfolgsrate nach Refactoring

Tabelle 4.3 gibt einen Überblick über die durchschnittliche Erfolgsrate nach Komplexität der Projekte und Programmiersprache für alle sechs untersuchten Refactoring-Aufgaben. Es fällt auf, dass zwischen den Refactoring-Aufgaben deutliche Unterschiede zu erkennen sind. Einfache Refactorings wie *Rename* oder *Inline Variable* erreichen sehr oft hohe Erfolgsraten. Insbesondere *Rename* wird in beiden Sprachen nahezu immer korrekt umgesetzt und erzielt eine durchschnittliche Erfolgsrate von 92,5%. Stärkere Schwankungen sind bei den mittleren Refactorings: *Getter-Setter* und *Guard Clauses* zu erkennen. Während *Getter-Setter*-Refactorings in Python-Projekten durchgehend erfolgreich sind, fallen die Erfolgsraten bei Apex teilweise deutlich niedriger aus. Dies könnte daran liegen, dass *Getter-Setter*-Refactorings in Python viel einfacher umzusetzen sind, da durch die Verwendung von `@property`-Dekoratoren die Zugriffe auf das jeweilige Attribut unverändert bleiben. Die niedrigsten Erfolgsraten treten bei den strukturell aufwendigeren Refactorings auf. Vor allem die *COC-Reduktion*, welche insgesamt am schlechtesten abgeschnitten hat, hatte nur eine Erfolgsrate von 46,67%. Vor allem ist deutlich zu erkennen, dass Unterschiede zwischen den einzelnen Refactoring-Typen einen stärkeren Einfluss auf die Erfolgsrate haben als die Komplexitätsstufe der betrachteten Projekte. Insgesamt lässt sich ableiten, dass die Umsetzung von strukturell komplexeren Refactorings deutlich häufiger zu Fehlern führt als die von einfacheren. Auffällig ist, dass die vorab definierte Schwierigkeitskategorie – einfach, mittel, schwierig – sich in der Erfolgsrate von *Gemini 3 Pro* nur leicht erkennbar macht, während sie für *Codestral-2501* einen klaren Trend zeigt. Das deutet darauf hin, dass *Gemini 3 Pro* auch bei konzeptionell anspruchsvollen Refactorings wie dem Strategy Pattern zuverlässig arbeitet, während *Codestral-2501* bei strukturellen oder semantischen Transformationen deutlich schwächer abschneidet. In Beachtung des Gesamtergebnisses, zeigt sich ebenfalls ein klarer Trend (vgl. Abbildung 4.3): Mit steigender Schwierigkeit der Refactoring-Typen nimmt die durchschnittliche Erfolgsrate ab. Während einfache Refactorings noch hohe Werte erreichen (*Rename*: 92,5%, *Inline Variable*: 84,2%), fällt das Gesamtergebnis bei mittleren Refactorings bereits deutlich ab (*Getter-Setter*: 83,3%,

Guard Clauses: 64,2%) und erreicht bei schwierigen Refactorings die niedrigsten Werte (*Strategy Pattern*: 64,2%, *COC-Reduktion*: 46,7%). Somit kann daraus gedeutet werden: Je schwieriger die Refactoring-Aufgabe, desto geringer ist die Erfolgsrate.

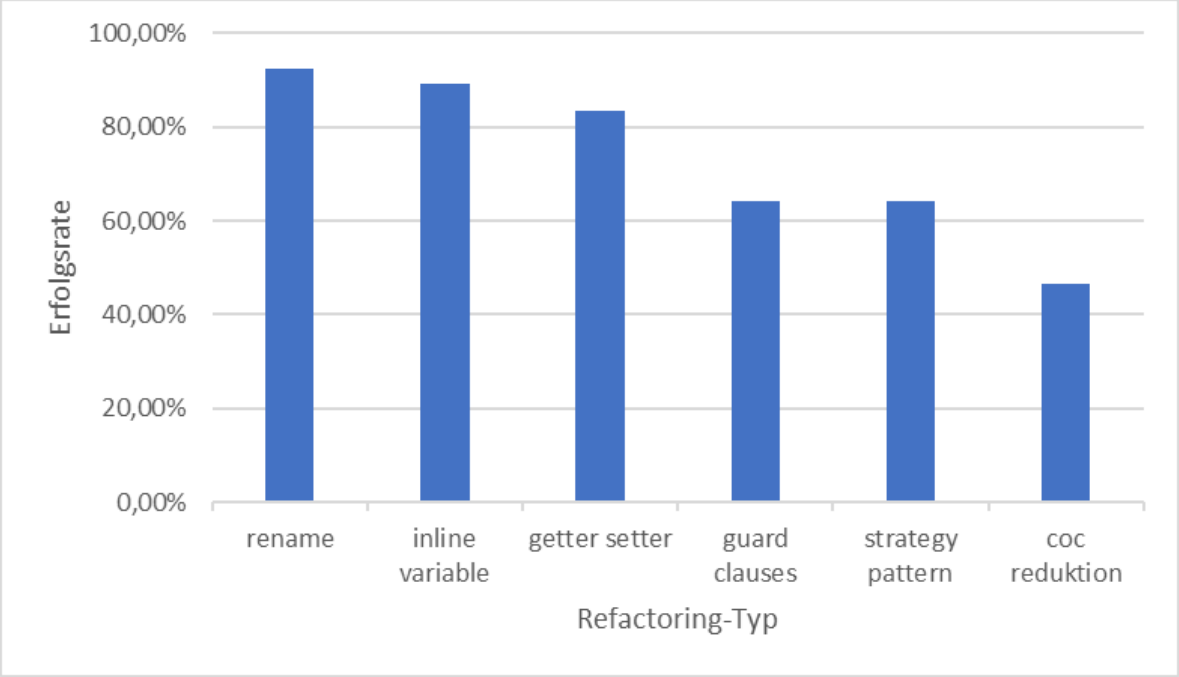

Abbildung 4.3.: Erfolgsrate nach Refactoring

Tabelle 4.3.: Erfolgsrate nach Refactoring, Programmiersprache und Komplexitätsstufe

	Apex				**Python**				**Gesamt**
Refactoring	**gering**	**mittel**	**komplex**	**Gesamt**	**gering**	**mittel**	**komplex**	**Gesamt**	
Rename	90,00% (n=20)	100,00% (n=20)	100,00% (n=20)	96,67% (n=60)	80,00% (n=20)	95,00% (n=20)	90,00% (n=20)	88,33% (n=60)	92,50% (n=120)
Inline Variable	95,00% (n=20)	75,00% (n=20)	95,00% (n=20)	88,33% (n=60)	80,00% (n=20)	90,00% (n=20)	100,00% (n=20)	90,00% (n=60)	89,17% (n=120)
Getter-Setter	50,00% (n=20)	100,00% (n=20)	50,00% (n=20)	66,67% (n=60)	100,00% (n=20)	100,00% (n=20)	100,00% (n=20)	100,00% (n=60)	83,33% (n=120)
Guard Clauses	65,00% (n=20)	65,00% (n=20)	30,00% (n=20)	53,33% (n=60)	75,00% (n=20)	50,00% (n=20)	100,00% (n=20)	75,00% (n=60)	64,17% (n=120)
Strategy Pattern	55,00% (n=20)	50,00% (n=20)	100,00% (n=20)	68,33% (n=60)	55,00% (n=20)	55,00% (n=20)	70,00% (n=20)	60,00% (n=60)	64,17% (n=120)
COC-Reduktion	55,00% (n=20)	65,00% (n=20)	15,00% (n=20)	45,00% (n=60)	50,00% (n=20)	45,00% (n=20)	50,00% (n=20)	48,33% (n=60)	46,67% (n=120)
Gesamtergebnis	68,33% (n=120)	75,83% (n=120)	65,00% (n=120)	69,72% (n=360)	73,33% (n=120)	72,50% (n=120)	85,00% (n=120)	76,94% (n=360)	73,33% (n=720)

n = Anzahl der einzelnen Refactoring-Ausführungen pro Zelle

4.3. Statistische Analyse zur Beantwortung von RQ1 und RQ2

Zur statistischen Überprüfung der zuvor beschriebenen Zusammenhänge wurde eine logistische Regressionsanalyse durchgeführt (Tabelle 4.4). Ziel der Analyse war es zu untersuchen, ob die beobachteten Unterschiede in den Erfolgsraten der Refactorings statistisch signifikant sind und welche Faktoren die Erfolgswahrscheinlichkeit beeinflussen. Als abhängige Variable wurde der Erfolg eines Refactorings modelliert. Für die kategorialen Variablen wurden Referenzkategorien definiert. Als Referenz für das verwendete Modell wurde *Codestral-2501* gewählt, für die Programmiersprache Apex und für die Projektkomplexität die Stufe „gering“. Der Intercept beschreibt somit die Erfolgswahrscheinlichkeit für Refactorings in Apex-Projekten geringer Komplexität unter Verwendung des Modells *Codestral-2501*. Die weiteren Koeffizienten geben jeweils die Veränderung der Erfolgswahrscheinlichkeit relativ zu dieser Basiskonstellation an.

Einfluss der Projektkomplexität (RQ1) Die Ergebnisse zeigen keinen signifikanten Unterschied zwischen Projekten mittlerer und geringer Komplexität ($p = 0{,}695$). Für Projekte hoher Komplexität ergibt sich hingegen ein signifikanter Effekt im Vergleich zur Referenzkategorie geringer Komplexität ($p = 0{,}019$). Dieser Effekt muss jedoch im Zusammenhang mit dem Interaktionseffekt zwischen Modell und Komplexität betrachtet werden.

Einfluss des verwendeten Modells Einen besonders starken Einfluss zeigt das verwendete Modell. Für *Gemini 3 Pro* ergibt sich ein hochsignifikanter Effekt ($p < 0{,}001$). Die geschätzte Odds Ratio von etwa 49,7 deutet darauf hin, dass die Wahrscheinlichkeit eines erfolgreichen Refactorings unter Verwendung dieses Modells deutlich höher ist als bei *Codestral-2501*.

Einfluss der Programmiersprache (RQ2) Auch die Programmiersprache zeigt einen signifikanten Einfluss auf die Erfolgswahrscheinlichkeit. Für Python ergibt sich im Vergleich zur Referenzsprache Apex eine signifikant höhere Erfolgswahrscheinlichkeit ($p = 0{,}003$). Die geschätzte Odds Ratio von etwa 1,77 bedeutet, dass erfolgreiche Refactorings in Python-Projekten etwa 1,77-mal wahrscheinlicher sind als in vergleichbaren Apex-Projekten. Dieses Ergebnis ist insbesondere für die Beantwortung von RQ2 relevant, da es darauf hindeutet, dass die untersuchten Modelle in der etablierten Programmiersprache Python insgesamt bessere Ergebnisse erzielen als in der weniger verbreiteten Sprache Apex.

Interaktionseffekt Besonders relevant ist zudem der signifikante Interaktionseffekt zwischen hoher Komplexität und dem verwendeten Modell ($p = 0{,}003$). Dieser zeigt, dass der Einfluss der Projektkomplexität nicht für beide Modelle gleich ist. Während *Codestral-2501* mit zunehmender Komplexität tendenziell bessere Ergebnisse erzielt, zeigt sich bei *Gemini 3 Pro* eine leichte Abnahme der Erfolgsrate bei komplexen Projekten.

Tabelle 4.4.: Logistische Regression zur Erklärung der Erfolgswahrscheinlichkeit

Variable	**Estimate**	**Odds Ratio**	**p-Wert**
Intercept	-0.490	0.61	0.020*
Komplexität (mittel)	0.103	1.11	0.695
Komplexität (komplex)	0.618	1.86	0.019*
LLM (*Gemini 3 Pro*)	3.907	49.73	<0.001***
Sprache (Python)	0.571	1.77	0.003**
Komplexität×LLM (mittel)	-0.400	0.67	0.626
Komplexität×LLM (komplex)	-2.091	0.12	0.003**

Signifikanzniveaus: *** $p < 0{,}001$, ** $p < 0{,}01$, * $p < 0{,}05$

Insgesamt zeigen die Ergebnisse der Regressionsanalyse, dass sowohl das verwendete Modell als auch die Programmiersprache einen signifikanten Einfluss auf die Erfolgswahrscheinlichkeit haben, während der Einfluss der Projektkomplexität vom jeweiligen Modell abhängt. Somit besteht kein einheitlicher Zusammenhang zwischen der Projektkomplexität und der Erfolgswahrscheinlichkeit.

4.4. Detaillierte Analyse der Ergebnisse

In diesem Abschnitt werden die Häufigkeit der erzeugten Codeänderungen, die Korrektheit der Refactoring-Umsetzung und die funktionale Korrektheit anhand der Testergebnisse untersucht. Darüber hinaus wird betrachtet, in welchem Zusammenhang diese Aspekte zueinander stehen.

4.4.1. Häufigkeit generierter Codeänderungen

Bisher wurden nur die einzelnen Erfolgsraten angesehen, ohne die Fehlversuche zu kategorisieren. In diesem Abschnitt wird dies nachgeholt, um einen differenzierteren Blick auf die Ergebnisse zu erhalten. In Tabelle 4.5 ist der durchschnittliche Anteil der Iterationen dargestellt, in denen Codeänderungen vorgenommen wurden. Als Codeänderung gilt jede Veränderung einer Codezeile, wobei Unterschiede in Leerzeichen und Zeilenumbrüchen nicht berücksichtigt werden. Auffällig ist, dass die Änderungsrate sehr hoch ist, jedoch bei Python-Projekten häufiger Iterationen ohne Änderungen auftreten als bei Apex-Projekten. In Tabelle 4.6 zeigt sich, dass insbesondere bei den Refactorings *Guard Clauses*, *Strategy Pattern* und *COC-Reduktion* vergleichsweise häufig keine Codeänderungen erzeugt wurden. Auch wird klar, dass bei den Refactorings *Strategy Pattern* und *Guard Clauses* ein erheblicher Anteil der gescheiterten Refactorings darauf beruht, dass am Code gar keine Änderungen vorgenommen wurden.

Tabelle 4.5.: Durchschnittlicher Anteil an Iterationen mit Codeänderungen nach Programmiersprache und Komplexitätsstufe

Sprache	Komplexitätsstufe	Anteil der Iterationen mit Codeänderungen
Apex	gering	98,33% (n=240)
Apex	mittel	99,17% (n=240)
Apex	komplex	97,50% (n=240)
Python	gering	90,00% (n=240)
Python	mittel	90,83% (n=240)
Python	komplex	93,33% (n=240)
Gesamt	gering	94,17% (n=480)
Gesamt	mittel	95,00% (n=480)
Gesamt	komplex	95,42% (n=480)
Gesamt (alle Durchläufe)		**94,86% (n=720)**

n = Anzahl der einzelnen Refactoring-Ausführungen pro Zelle

Tabelle 4.6.: Anteil fehlgeschlagener Refactoring-Ausführungen nach Fehlerursache und Refactoring-Typ

Refactoring	Keine Codeänderung erzeugt	Anteil der Fehlrefactorings wegen keiner Änderung
Rename	0,00% (n=120)	0,00%
Inline Variable	3,33% (n=120)	100,00%
Getter/Setter	0,00% (n=120)	0,00%
Guard Clauses	15,00% (n=120)	43,90%
Strategy Pattern	5,83% (n=120)	77,78%
COC-Reduktion	6,67% (n=120)	15,38%
Gesamt	5,14% (n=720)	32,74%

n = Anzahl der einzelnen Refactoring-Ausführungen pro Zelle

4.4.2. Korrektheit der Refactoring-Umsetzung

Aus Tabelle 4.7 geht hervor, dass in 84% der Fälle Refactoring-Aufgaben richtig umgesetzt wurden. Doch allein das richtige Umsetzen eines Refactorings garantiert noch nicht, dass alle Tests bestehen. Wenn die einzelnen Refactoring-Typen betrachtet werden, zeigen sich deutliche Unterschiede. Besonders hohe Werte erreichen *Rename* mit 98,33%, *Getter-Setter* mit 95,83% und *Strategy Pattern* mit 92,50%. Demgegenüber fallen die Werte für *Guard Clauses* mit 65,83% und insbesondere für die *COC-Reduktion*

mit 56,67% deutlich geringer aus. Diese beiden Refactoring-Typen stellen damit die größte Herausforderung für die Modelle dar.

Tabelle 4.7.: Anteil korrekt umgesetzter Refactorings nach Refactoring-Typ, Programmiersprache und Komplexitätsstufe

	Apex				Python				
Refactoring	gering	mittel	komplex	Gesamt	gering	mittel	komplex	Gesamt	Gesamt
Rename	90,00% (n=20)	100,00% (n=20)	100,00% (n=20)	96,67% (n=60)	100,00% (n=20)	100,00% (n=20)	100,00% (n=20)	100,00% (n=60)	98,33% (n=120)
Inline Variable	95,00% (n=20)	100,00% (n=20)	95,00% (n=20)	96,67% (n=60)	100,00% (n=20)	90,00% (n=20)	100,00% (n=20)	96,67% (n=60)	96,67% (n=120)
Getter/Setter	75,00% (n=20)	100,00% (n=20)	100,00% (n=20)	91,67% (n=60)	100,00% (n=20)	100,00% (n=20)	100,00% (n=20)	100,00% (n=60)	95,83% (n=120)
Guard Clauses	65,00% (n=20)	65,00% (n=20)	40,00% (n=20)	56,67% (n=60)	75,00% (n=20)	50,00% (n=20)	100,00% (n=20)	75,00% (n=60)	65,83% (n=120)
Strategy Pattern	100,00% (n=20)	100,00% (n=20)	100,00% (n=20)	100,00% (n=60)	55,00% (n=20)	100,00% (n=20)	100,00% (n=20)	85,00% (n=60)	92,50% (n=120)
COC-Reduktion	55,00% (n=20)	100,00% (n=20)	25,00% (n=20)	60,00% (n=60)	50,00% (n=20)	60,00% (n=20)	50,00% (n=20)	53,33% (n=60)	56,67% (n=120)
Gesamt	80,00% (n=120)	94,17% (n=120)	76,67% (n=120)	83,61% (n=360)	80,00% (n=120)	83,33% (n=120)	91,67% (n=120)	85,00% (n=360)	84,31% (n=720)

n = Anzahl der einzelnen Refactoring-Ausführungen pro Zelle

4.4.3. Funktionale Korrektheit basierend auf Testergebnissen

Wie Tabelle 4.8 zeigt, weisen Python-Projekte im Durchschnitt eine höhere Testerfolgsrate auf als Apex-Projekte. Dieser Befund ist insofern bemerkenswert, als dass die Python-Projekte sowohl eine höhere absolute Testanzahl als auch eine höhere Testabdeckung aufweisen (vgl. Tabelle 3.4). Auffällig ist zudem, dass das Refactoring der *Guard Clauses* die höchste Test-Erfolgsrate aufweist, obwohl es bei der korrekten Umsetzung nur vergleichsweise geringe Werte erzielt.

4.4.4. Zusammenhang zwischen Refactoring-Korrektheit und funktionaler Korrektheit

Die getrennte Betrachtung von Refactoring-Korrektheit und Test-Erfolg legt nahe, dass beide Maße nicht vollständig deckungsgleich sind. Um diesen Zusammenhang genauer zu untersuchen, wurden beide Dimensionen gemeinsam ausgewertet. Tabelle 4.9 zeigt, wie häufig korrekt bzw. nicht korrekt umgesetzte Refactorings mit bestandenen bzw. nicht bestandenen Tests zusammenfielen. Es zeigt sich, dass in 528 Fällen sowohl das Refactoring korrekt umgesetzt, als auch alle Tests bestanden wurden. Gleichzeitig traten jedoch 79 Fälle auf, in denen das Refactoring als korrekt bewertet wurde, obwohl Tests fehlschlugen. Umgekehrt bestanden in 69 Fällen alle Tests, obwohl das Refactoring nicht korrekt umgesetzt worden war. Dies verdeutlicht, dass funktionale Korrektheit und Refactoring-Korrektheit zwar eng zusammenhängen, jedoch unterschiedliche Aspekte der Qualität erfassen. Ein Beispiel für den Fall, dass die Tests bestanden

Tabelle 4.8.: Anteil der Iterationen, in denen alle Tests erfolgreich bestanden wurden, nach Refactoring-Typ, Programmiersprache und Komplexitätsstufe

	Apex				Python				
Refactoring	gering	mittel	komplex	Gesamt	gering	mittel	komplex	Gesamt	Gesamt
Rename	95,00% (n=20)	100,00% (n=20)	100,00% (n=20)	98,33% (n=60)	80,00% (n=20)	95,00% (n=20)	90,00% (n=20)	88,33% (n=60)	93,33% (n=120)
Inline Variable	100,00% (n=20)	50,00% (n=20)	100,00% (n=20)	83,33% (n=60)	80,00% (n=20)	100,00% (n=20)	100,00% (n=20)	93,33% (n=60)	88,33% (n=120)
Getter/Setter	50,00% (n=20)	100,00% (n=20)	50,00% (n=20)	66,67% (n=60)	100,00% (n=20)	100,00% (n=20)	100,00% (n=20)	100,00% (n=60)	83,33% (n=120)
Guard Clauses	100,00% (n=20)	100,00% (n=20)	85,00% (n=20)	95,00% (n=60)	100,00% (n=20)	100,00% (n=20)	100,00% (n=20)	100,00% (n=60)	97,50% (n=120)
Strategy Pattern	55,00% (n=20)	50,00% (n=20)	100,00% (n=20)	68,33% (n=60)	100,00% (n=20)	55,00% (n=20)	70,00% (n=20)	75,00% (n=60)	71,67% (n=120)
COC-Reduktion	55,00% (n=20)	65,00% (n=20)	30,00% (n=20)	50,00% (n=60)	70,00% (n=20)	60,00% (n=20)	100,00% (n=20)	76,67% (n=60)	63,33% (n=120)
Gesamt	75,83% (n=120)	77,50% (n=120)	77,50% (n=120)	76,94% (n=360)	88,33% (n=120)	85,00% (n=120)	93,33% (n=120)	88,89% (n=360)	82,92% (n=720)

n = Anzahl der einzelnen Refactoring-Ausführungen pro Zelle

Tabelle 4.9.: Zusammenhang zwischen Refactoring-Korrektheit und funktionaler Korrektheit

	Tests bestanden	Tests fehlgeschlagen	Gesamt
Refactoring korrekt umgesetzt	528 (73,33%)	79 (10,97%)	607 (84,31%)
Refactoring nicht korrekt umgesetzt	69 (9,58%)	44 (6,11%)	113 (15,69%)
Gesamt	597 (82,92%)	123 (17,08%)	720 (100,00%)

wurden, obwohl das Refactoring falsch bzw. unvollständig umgesetzt wurde, ist in Listing 4.6 dargestellt. Der umgekehrte Fall, bei dem das beabsichtigte Refactoring korrekt umgesetzt wurde und die resultierende Änderung zu fehlgeschlagenen Tests führte, trat ebenfalls auf. Ein entsprechendes Beispiel ist beim Inline-Variable-Refactoring zu beobachten. Dabei wurde eine temporäre Variable korrekt durch ihren direkten Ausdruck ersetzt, sodass das Refactoring als erfolgreich umgesetzt bewertet werden konnte (vgl. Listing 4.1). Dennoch schlug der Test fehl, da zusätzlich eine Rückgabe von `mode.value` eingeführt wurde (vgl. Listing 4.2). Da im Test jedoch ein Integer übergeben wurde, führte dies zu einer `AttributeError`-Exception. Das Beispiel zeigt, dass ein korrekt umgesetztes Refactoring dennoch mit funktionalen Fehlern verbunden sein kann, wenn darüber hinaus zusätzliche Änderungen am Verhalten entstehen.

Listing 4.1.: Korrekt umgesetztes Inline-Variable-Refactoring mit zusätzlicher problematischer Änderung

```
--- backup/win32.py
+++ refactored/win32.py
@@ -105,2 +105 @@
-handle = _GetStdHandle(stream_id)
-return _SetConsoleTextAttribute(handle, attrs)
+return _SetConsoleTextAttribute(_GetStdHandle(stream_id), attrs)
@@ -144,0 +144 @@
+return mode.value
```

Listing 4.2.: Fehlschlagender Test nach korrekt umgesetztem Inline-Variable-Refactoring

```
> fake_win32.SetConsoleMode(123, 1)

E AttributeError: 'int' object has no attribute 'value'
```

4.5. Analyse der Fehlertypen

In diesem Abschnitt werden die typischen Fehlermuster analysiert. Diese lassen sich grob in fünf Kategorien einteilen: keine Codeänderung, reine Umformulierungen, unvollständige Umsetzung, Änderungen an nicht gefragten Stellen sowie die Erhöhung der COC durch zusätzliche `if`-Abfragen. Pro Refactoring-Typ wird ein repräsentatives Negativbeispiel anhand relevanter Codeausschnitte illustriert, vollständigere Ausschnitte befinden sich im Anhang.

Rename Aus dem Rename-Refactoring sind zwei typische Fehlermuster entstanden. Einerseits kam es häufiger vor, dass eine Methode umbenannt wurde, die nicht umbenannt werden sollte. Darüber hinaus kam es häufig vor, dass eine Referenz zur Umbenennung nicht aktualisiert wurde. Das nachfolgende Beispiel veranschaulicht den ersten Fall: Es wurde nur die Umbenennung der Methode `should_wrap` gefordert, aber stattdessen erfolgte eine Umbenennung der `convert`-Variablen.

Listing 4.3.: Aufgabenbeschreibung zum *Rename*-Refactoring bei *Colorama*

```
Your Task:
1. Rename the method `should_wrap` in the file: `ansitowin32.py` to `is_wrappable`
2. Update every relevant occurrence across the entire project.
3. For EACH modified file, respond exactly in the following format:
```

Listing 4.4.: Originalcode-Abschnitt bei fehlerhaftem *Rename*-Refactoring

```
if new_stdout.convert:
    sys.stdout = new_stdout
new_stderr = AnsiToWin32(sys.stderr, convert=None, strip=None, autoreset=False)
if new_stderr.convert:
    sys.stderr = new_stderr
```

Listing 4.5.: Transformierter Abschnitt bei fehlerhaftem *Rename*-Refactoring

```
if new_stdout.is_wrappable():
    sys.stdout = new_stdout
new_stderr = AnsiToWin32(sys.stderr, convert=None, strip=None, autoreset=False)
if new_stderr.is_wrappable():
    sys.stderr = new_stderr
```

Inline Variable Bei diesem Refactoring trat ausschließlich das Muster auf, dass keine Codeänderung vorgenommen wurde. Fehlerhaft umgesetzte Transformationen kamen nicht vor, weshalb kein Codebeispiel dargestellt wird.

Getter-Setter Das häufigste Fehlermuster bestand darin, dass nur ein Getter erzeugt, der zugehörige Setter jedoch weggelassen wurde. In Apex-Projekten kam zusätzlich vor, dass zwar beide Methoden erstellt wurden, die bestehenden Zugriffe im restlichen Code aber nicht auf die neue Getter-Setter-Schnittstelle umgestellt wurden. Das folgende

Beispiel zeigt den ersten Fall, die Klasse enthält lediglich `getIdFieldName()`, ein entsprechender Setter fehlt vollständig.

Listing 4.6.: Relevanter transformierter Abschnitt beim fehlerhaften Getter-Setter-Refactoring

```
1  public with sharing class Relationship {
2      ...
3      private final String idFieldName;
4      private Relationship(String referenceFieldName, String idFieldName) {
5          this.idFieldName = idFieldName;
6      }
7      ...
8      public String getIdFieldName() {
9          return idFieldName;
10     }
11 }
```

Guard Clauses Bei diesem Refactoring wurde oft keine Änderung vorgenommen. Wenn eine Transformation stattfand, wurde oft nur die Reihenfolge der `if`-Statements geändert, ohne die Verschachtelungstiefe zu verringern oder Early Exits hinzuzufügen. In manchen Fällen war die Anzahl der verschachtelten `if`-Abfragen nach dem Refactoring sogar höher als zuvor. Das nachfolgende Beispiel zeigt diesen Fall. Der Originalcode hatte einen zentralen `if`-Block mit zwei inneren Zusatzprüfungen. Im transformierten Code wurden mehrere separate `if`-Blöcke mit eigenen inneren Abfragen erstellt, was die Verschachtelung insgesamt erhöhte anstatt sie zu reduzieren. Der transformierte Abschnitt in vollem Umfang ist im Anhang zu finden (vgl. Abschnitt A.4).

Listing 4.7.: Ausschnitt aus der Methode `getRepoFromSObjectType` vor dem Guard-Clauses-Refactoring

```
...
if (
  queriedResults.size() > 0 ||
  aggRecords?.size() > 0 ||
  historyRecords?.size() > 0 ||
  cursorResults?.size() > 0 ||
  alwaysUseMock == true
) {
  RepoMock mock = new RepoMock(sObjectType, repoFactory);
  mock.results.addAll(queriedResults);
  if (aggRecords != null) {
    mock.aggRecords.addAll(aggRecords);
  }
  if (historyRecords != null) {
    mock.historyRecords.addAll(historyRecords);
  }
  repo = mock;
} else {
  repo = (IHistoryRepository) fallback;
}
...
```

Strategy Pattern Auch hier war keine Codeänderung das häufigste Fehlermuster. Darüber hinaus kam vor, dass zwar die Struktur des Strategy-Patterns implementiert, diese aber nicht verwendet wurde. Somit war hier der Fall, dass zwar funktionell alles korrekt wie vorher ausgeführt, aber die alte Methode unbearbeitet gelassen wurde. Somit wurde gar nicht auf das *Strategy Pattern* zugegriffen und das Refactoring wurde als nicht korrekt markiert. In diesem Beispiel wird genau dieser Fall aufgeschlüsselt. Die neu eingeführte Implementierung wird nicht verwendet. Obwohl alle Tests bestanden, wurde das *Strategy Pattern* somit nicht korrekt umgesetzt. Ein gekürzter Ausschnitt des transformierten Codes ist im Anhang dargestellt (vgl. Abschnitt A.5).

Listing 4.8.: Transformierter Abschnitt der Methode `call_win32`

```
...
def call_win32(self, command, params):
    if command == 'm':
        for param in params:
            if param in self.win32_calls:
                func_args = self.win32_calls[param]
                func = func_args[0]
                args = func_args[1:]
                kwargs = dict(on_stderr=self.on_stderr)
                func(*args, **kwargs)
    elif command in 'J':
        winterm.erase_screen(params[0], on_stderr=self.on_stderr)
    elif command in 'K':
        winterm.erase_line(params[0], on_stderr=self.on_stderr)
    elif command in 'Hf':
        winterm.set_cursor_position(params, on_stderr=self.on_stderr)
    elif command in 'ABCD':
        n = params[0]
        x, y = {'A': (0, -n), 'B': (0, n), 'C': (n, 0), 'D': (-n, 0)}[command]
        winterm.cursor_adjust(x, y, on_stderr=self.on_stderr)
...
```

COC-Reduktion Typische Fehler bestanden darin, keine Änderungen vorzunehmen oder lediglich einfache, reine Umformulierungen vorzunehmen, die jedoch keinen Einfluss auf die COC hatten. Allerdings kam es bei zahlreichen Refactorings von Apex-Projekten vor, dass die COC stark anstieg. Der Grund hierfür war die große Anzahl an hinzugefügten Early Exits. Diese frühen Ausstiege bestanden aus `if`-Anweisungen, und die COC steigt mit der Anzahl der `if`-Anweisungen. In diesem Beispiel wurde das gesamte Projekt um eine COC in Höhe von 1555 angehoben. Zur besseren Übersicht ist ein Ausschnitt der Änderungsdatei dargestellt (vgl. Listing 4.9). In dieser ist deutlich zu sehen, dass lediglich `if`-Statements ergänzt wurden, was zur Folge hatte, dass sich die COC derart gesteigert hat. Ein etwas größerer Ausschnitt der Codeänderungen ist im Anhang dargestellt (vgl. Abschnitt A.6).

Listing 4.9.: Gekürzte Version der Veränderung im Refactoring: *COC-Reduktion*

```
...
--- backup/repository\Repository.cls
+++ refactored/repository\Repository.cls
@@ -20,0 +21,3 @@
+if(repoType==null||queryFields==null||repoFactory==null){
+thrownewIllegalArgumentException('Repositoryparameterscannotbenull');
+}
@@ -26,0 +30,3 @@
+if(queries==null){
+queries=newList<Query>();
+}
...
```

4.6. Stabilität der Ergebnisse

Da jedes Refactoring pro Modell und Projekt zehnmal ausgeführt wurde, lässt sich neben der durchschnittlichen Erfolgsrate auch die Stabilität der Ergebnisse untersuchen. Ein Modell mit geringer Streuung liefert über mehrere Durchläufe hinweg konsistente Ergebnisse, während eine hohe Streuung darauf hinweist, dass identische Eingaben zu stark variierenden Resultaten führen können. Tabelle 4.10 zeigt die durchschnittliche Standardabweichung der Erfolgsraten, aufgeschlüsselt nach Modell und Programmiersprache.

Tabelle 4.10.: Standardabweichung der Erfolgsraten nach Modell und Programmiersprache

Modell	**Apex**	**Python**	**Gesamt**
Codestral-2501	41,73%	39,97%	40,63%
Gemini 3 Pro	19,65%	10,29%	15,58%
Gesamt	39,75%	35,04%	37,38%

Es zeigt sich, dass *Gemini 3 Pro* in beiden Programmiersprachen eine deutlich geringere Standardabweichung aufweist als *Codestral-2501*. Während die durchschnittliche Streuung bei *Gemini 3 Pro* insgesamt bei 15,58% liegt, erreicht *Codestral-2501* eine Standardabweichung von 40,63%. Dies deutet darauf hin, dass die Ergebnisse von *Gemini 3 Pro* über mehrere Iterationen hinweg konsistenter sind, während *Codestral-2501* deutlich stärkere Schwankungen zeigt. Auch zwischen den Programmiersprachen lassen sich Unterschiede erkennen: Insgesamt fällt die Streuung mit *Codestral-2501* in Python mit 35,04% etwas geringer aus als in Apex mit 39,75%. Neben der modellabhängigen Stabilität lässt sich auch untersuchen, wie stark die Ergebnisse je nach Refactoring-Typ und Komplexitätsstufe variieren. Tabelle 4.11 zeigt die Standardabweichung der

Erfolgsraten für die einzelnen Refactoring-Typen.

Tabelle 4.11.: Standardabweichung der Erfolgsraten nach Refactoring-Typ und Komplexitätsstufe

Refactoring	**gering**	**mittel**	**komplex**	**Gesamt**
Rename	19,15%	5,00%	10,00%	12,88%
Inline Variable	9,57%	23,63%	5,00%	15,05%
Getter/Setter	50,00%	0,00%	50,00%	38,92%
Guard Clauses	29,44%	50,58%	43,59%	38,48%
Strategy Pattern	51,96%	55,00%	30,00%	45,22%
COC-Reduktion	55,00%	33,17%	47,17%	42,92%
Gesamt	37,87%	36,94%	38,79%	37,38%

Die Ergebnisse zeigen deutliche Unterschiede zwischen den Refactoring-Typen. Besonders geringe Streuungen treten beim Refactoring *Rename* auf, das mit einer Gesamtstandardabweichung von 12,88% vergleichsweise stabile Ergebnisse liefert. Deutlich höhere Streuungen zeigen hingegen komplexere Refactorings wie *Strategy Pattern* (45,22%) oder die *COC-Reduktion* (42,92%). Dies deutet darauf hin, dass solche Aufgaben für die Modelle schwerer konsistent umzusetzen sind. Die Stabilität zeigt sich über die verschiedenen Komplexitätsstufen hinweg als weitgehend konstant. Während einfache Projekte im Durchschnitt eine Standardabweichung von 37,87% aufweisen, liegt diese bei mittlerer Komplexität bei 36,94% und bei komplexen Projekten bei 38,79%. Zusammenfassend lässt sich feststellen, dass *Gemini 3 Pro* insgesamt stabilere Ergebnisse liefert als *Codestral-2501*. Gleichzeitig zeigen die Ergebnisse, dass insbesondere komplexere Refactoring-Aufgaben zu deutlich größeren Schwankungen führen können, selbst wenn identische Prompts und Projekte verwendet werden. Dies verdeutlicht, dass neben der durchschnittlichen Erfolgsrate auch die Stabilität der Ergebnisse ein wichtiger Faktor für den praktischen Einsatz von LLMs in automatisierten Refactoring-Szenarien ist.

4.7. Zusammenfassung der Ergebnisse

Die zentralen Ergebnisse der Untersuchung sind in Tabelle 4.12 zusammengefasst. Die Tabelle stellt die wichtigsten Kennzahlen der Studie entlang der Komplexitätsstufen sowie zentraler Prozessmetriken dar und ermöglicht damit einen kompakten Überblick über die beobachteten Zusammenhänge. Zunächst zeigt sich, dass die Programmiersprache einen leichten Einfluss auf die Erfolgsraten der Refactorings hat. Über alle Komplexitätsstufen hinweg erreichen Python-Projekte höhere Erfolgsraten als Apex-Projekte. Während Apex insgesamt eine Erfolgsrate von 69,7 % erzielt, liegt die Erfolgsrate in Python bei 76,9 %. Besonders deutlich wird dieser Unterschied bei komplexen Projekten, in denen Python mit 85,0 % eine deutlich höhere Erfolgsrate erreicht als Apex mit 65,0 %.

Im Gegensatz dazu zeigt sich kein klarer Zusammenhang zwischen der Projektkomplexität und der Erfolgsrate. Wie in Tabelle 4.12 zu erkennen ist, bleiben die durchschnittlichen Erfolgsraten über alle drei Komplexitätsstufen hinweg relativ stabil (70,8 % bei geringer, 74,2 % bei mittlerer und 75,0 % bei hoher Komplexität). Die Tabelle verdeutlicht zudem, dass die Modelle in den meisten Iterationen tatsächlich Codeänderungen generieren. In insgesamt 94,86% der Fälle wurde eine Änderung am Code vorgenommen. Davon wurden 84,31% der Refactorings korrekt umgesetzt, während in 82,92% der Fälle anschließend auch alle Tests bestanden wurden. Diese Werte zeigen, dass ein Großteil der Fehlversuche nicht darauf zurückzuführen ist, dass keine Änderungen vorgenommen wurden, sondern vielmehr auf fehlerhafte Implementierungen der Refactorings oder auf Änderungen, die bestehende Tests brechen. Ein weiterer wichtiger Einflussfaktor ist der Refactoring-Typ. Einfache Refactorings wie *Rename* oder *Getter/Setter* werden von den Modellen mit sehr hohen Erfolgsraten umgesetzt, während komplexere Aufgaben wie *Strategy Pattern* oder insbesondere die *COC-Reduktion* deutlich häufiger zu Fehlern führen. Diese Aufgaben erfordern in der Regel umfangreichere strukturelle Änderungen am Code, was die Wahrscheinlichkeit fehlerhafter Implementierungen erhöht. Schließlich zeigt Tabelle 4.12 auch deutliche Unterschiede zwischen den untersuchten Modellen. Während *Gemini 3 Pro* über alle Kategorien hinweg sehr hohe Erfolgsraten erzielt (94,7 %) und vergleichsweise stabile Ergebnisse liefert, erreicht *Codestral-2501* mit 51,9 % deutlich niedrigere Erfolgsraten. Diese Unterschiede treten konsistent über verschiedene Programmiersprachen, Komplexitätsstufen und Refactoring-Typen hinweg auf.

Tabelle 4.12.: Überblick zentraler Ergebnisse nach Komplexitätsstufe

Kennzahl	**gering**	**mittel**	**komplex**	**Gesamt**
Erfolgsraten				
Apex	68,3 %	75,8 %	65,0 %	69,7 %
Python	73,3 %	72,5 %	85,0 %	76,9 %
Gesamte Erfolgsrate	70,8 %	74,2 %	75,0 %	73,3 %
Prozessmetriken				
Änderung vorgenommen	94,2 %	95,0 %	95,4 %	94,9 %
Refactoring korrekt umgesetzt	80,0 %	94,2 %	76,7 %	84,3 %
Tests bestanden	82,1 %	81,3 %	85,4 %	82,9 %
Modelle				
Codestral-2501	44,2 %	51,7 %	60,0 %	51,9 %
Gemini 3 Pro	97,5 %	96,7 %	90,0 %	94,7 %

Weitere Kennzahlen	**Ergebnis**
Bestes Refactoring	Rename (92,5 % Erfolgsrate)
Schwächstes Refactoring	COC-Reduktion (46,7 %)
Höchste Stabilität	Rename (12,9 % Standardabweichung)
Geringste Stabilität	Strategy Pattern (45,2 % Standardabweichung)
Totalausfälle *Codestral-2501*	8 von 36 Kombinationen (22,2 %)
Totalausfälle *Gemini 3 Pro*	0 von 36 Kombinationen (0,0 %)
Durchschnittliche Standardabweichung	37,4 %

Totalausfall: Keine erfolgreiche Refactoring-Ausführung in allen zehn Iterationen einer Kombination aus Projekt, Refactoring und LLM.

5. Diskussion

In diesem Kapitel werden die zuvor dargestellten Ergebnisse interpretiert und in einen größeren fachlichen Zusammenhang eingeordnet. Dazu werden die zentralen Befunde im Hinblick auf die Forschungsfragen diskutiert und mit bestehenden Arbeiten aus der Literatur verglichen. Darüber hinaus werden methodische Einschränkungen der Untersuchung reflektiert und die praktischen Implikationen der Ergebnisse für den Einsatz von LLMs bei Refactorings herausgearbeitet. Abschließend werden unerwartete Befunde aufgegriffen, um ein möglichst vollständiges Bild der Untersuchungsergebnisse zu geben.

5.1. Interpretation der Ergebnisse

Dieses Kapitel ordnet die Ergebnisse aus Kapitel 4 ein und diskutiert sie im Kontext der Forschungsfragen. Im Fokus stehen zwei Punkte: der Einfluss von COC auf den Erfolg von LLM-basierten Refactorings (RQ1) und Unterschiede zwischen einer etablierten Sprache (Python) und einer weniger verbreiteten Sprache (Apex) (RQ2). Der auffälligste Befund ist die stark abweichende durchschnittliche Erfolgsrate der beiden genutzten LLMs. Somit ist auch die Standardabweichung über Iterationen bei *Gemini 3 Pro* wesentlich geringer. Eine mögliche Erklärung für diese Unterschiede liegt in der Architektur und dem Trainingsfokus der Modelle. Während *Codestral-2501* speziell auf Codegenerierung optimiert wurde [25], handelt es sich bei *Gemini 3 Pro* um ein sehr großes multimodales Modell mit breiteren Trainingsdaten [1]. Dies könnte dazu führen, dass *Gemini 3 Pro* robuster gegenüber unterschiedlichen Refactoringaufgaben ist. Außerdem ist *Gemini 3 Pro* am 19.11.2025 erschienen [41]. Dies ist fast ein Jahr später als das Erscheinungsdatum von *Codestral-2501* am 25.01.2025 [26]. Angesichts der schnellen Entwicklung im Bereich der LLM-Forschung stellen 11 Monate einen erheblichen Zeitraum dar, in dem sich sowohl Modelle als auch Benchmarks substanziell weiterentwickelt haben. [64, 69]. Außerdem ist aus den Ergebnissen (vgl. Tabelle 4.1) nicht ersichtlich, dass die COC einen Einfluss auf die Qualität der Ergebnisse haben könnte. Der viel dominantere Faktor ist die Auswahl der Refactorings, was vor allem bei *Codestral-2501* sichtbar ist. Bei *Gemini 3 Pro* ist dieser Trend auch erkennbar, jedoch nicht so eindeutig, weil das LLM stets mit fast allen Aufgaben sehr gut zurecht kommt (vgl. Tabelle 4.1). Bei beiden LLMs ist zu erkennen, dass die Programmiersprache einen leichten Einfluss auf die Erfolgsrate hat. Hierbei ist in Abbildung 4.2 zu erkennen, dass *Gemini 3 Pro* einen 3,9%-tigen und *Codestral-2501* einen 13,9% Anstieg verweisen. Ob dieser Anstieg von der Programmiersprache abhängt, lässt sich aufgrund der limitierten Ergebnisse nicht klar sagen.

5.2. Vergleich mit verwandten Arbeiten

Im Vergleich zu bestehenden Studien fällt zunächst auf, dass viele Arbeiten LLM-basierte Refactorings entweder auf nur weitverbreitete Sprachen fokussieren oder andere Zielmetriken nutzen [81, 24]. So wurde in dieser Arbeit primär die COC untersucht, während Shirafuji u. a. [81] die CC untersuchte. Im Unterschied dazu adressiert diese Arbeit zwei Aspekte gemeinsam: reale Projekte unterschiedlicher COC, sowie einen Sprachvergleich mit Apex als weniger verbreiteter Sprache. Zusätzlich wird nicht nur funktionale Korrektheit über Tests betrachtet, sondern auch die korrekte Umsetzung des jeweiligen Refactorings (je nach Typ manuell bzw. über spezifische Validierung). Dadurch sind die Ergebnisse stärker an einem praktischen Refactoring-Szenario ausgerichtet, in dem es nicht genügt, dass Tests bestanden wurden, sondern das Refactoring auch tatsächlich wie gefordert umgesetzt sein muss. Vergleicht man die Ergebnisse zwischen den beiden LLMs, dann folgen sie demselben Trend wie dem *Aider Polyglot* Benchmark [3]. In diesem Benchmark wurden 225 unterschiedliche Programmieraufgaben in den Programmiersprachen C++, Go, Java, JavaScript, Python und Rust durchgeführt. Zu den evaluierten Modellen zählte unter anderem *Gemini 2.5 Pro*. Google positioniert *Gemini 3 Pro* als Nachfolgemodell innerhalb der Gemini-3-Serie und hebt hervor, dass es *Gemini 2.5 Pro* in jeden wichtigen Benchmark übertrifft [1]. Dies spricht dafür, dass auch im *Aider Polyglot Benchmark* mindestens vergleichbare oder tendenziell bessere Ergebnisse zu erwarten gewesen wären, ohne dass sich dies daraus unmittelbar beweisen lässt. Ebenfalls untersucht wurde *Codestral-2501*, das auch in der vorliegenden Arbeit verwendet wird. *Gemini 2.5 Pro* erhielt eine 83,1%-ige Erfolgsrate, während *Codestral-2501* nur eine Erfolgsrate von 11,1% hatte. Da *Gemini 3 Pro* auf *Gemini 2.5 Pro* aufbaut, war eine ähnlich starke Performance zu erwarten. Tatsächlich zeigen auch die Ergebnisse dieser Arbeit das gleiche Muster: *Codestral-2501* liegt deutlich hinter *Gemini 3 Pro* zurück.

5.3. Gefährdung der Validität

Die diskutierten Ergebnisse liefern zwar wichtige Hinweise auf die Leistungsfähigkeit von LLMs beim Refactoring, sind jedoch nicht ohne Weiteres verallgemeinerbar. Für eine angemessene Einordnung der Befunde ist es daher erforderlich, die methodischen und praktischen Limitationen der Untersuchung zu reflektieren.

5.3.1. Kleine Stichprobe und Generalisierbarkeit

Das vorliegende Experiment basiert auf nur sechs Open-Source-Projekten (drei Python, drei Apex), was die statistische Aussagekraft und Generalisierbarkeit einschränkt. Eine Post-hoc-Poweranalyse ergab einen beobachteten Effekt von $d = 0,335$, ein Maß dafür, wie stark sich die Erfolgsraten zwischen den Komplexitätsstufen unterscheiden, sowie eine statistische Power von lediglich 10,3%, demnach hatte die Arbeit nur eine Wahrscheinlichkeit von 10,3%, einen tatsächlich vorhandenen Effekt zu entdecken. Basierend auf diesen Werten zeigt eine anschließende A-priori-Poweranalyse [36], dass bei einem

Signifikanzniveau von $\alpha = 0,05$, der maximal tolerierten Wahrscheinlichkeit für einen fälschlichen Befund und einer angestrebten Power von 0,80 mindestens 72 Projekte erforderlich gewesen wären. Obwohl die Projektauswahl systematisch über das Verhältnis $\frac{\text{LOC}}{\text{COC}}$ erfolgte, um Größenunterschiede zu minimieren, können andere Codebasen, wie etwa aus Web-Entwicklung, Embedded Systems oder Enterprise-Software, abweichende Verhaltensmuster zeigen. Zudem wurden ausschließlich zwei LLMs (*Gemini 3 Pro*, *Codestral-2501*) verglichen. Neuere Modelle wie GPT-5.4, *Gemini 3.1 Pro* oder auf Code spezialisierte Varianten könnten andere Ergebnisse liefern und die beobachteten Trends relativieren.

5.3.2. Testqualität und Validierungsprobleme

Die Testinfrastruktur bildet eine weitere zentrale Limitation. Zwar wurde eine einheitliche Testanzahl angestrebt (426 Python, 129 Apex), doch decken die Testfälle nicht alle Randfälle ab. Besonders problematisch: Generierte Tests (bis zu 90% bei *colorama*) können unvollständige oder fehlerhafte Abdeckung bieten. Bei Apex-Projekten scheiterten Copilot-generierte Tests häufig beim Deployment oder liefen dauerhaft fehlerhaft, weshalb die Abdeckung bei 84–88% stagnierte. Darüber hinaus unterscheiden sich die Projekte in der absoluten Anzahl der vorhandenen Tests. Obwohl eine vergleichbare Anzahl der Testfälle und Testabdeckung angestrebt wurde, kann eine unterschiedliche Anzahl an Tests dazu führen, dass Änderungen in manchen Projekten stärker oder schwächer erkannt werden als in anderen. Die manuelle Validierung von strukturellen Refactorings wie *Strategy Pattern* und *Guard Clauses* unterliegt zudem Subjektivität, trotz definierter Kriterien (vgl. Unterabschnitt 3.6.2). Da die manuelle Bewertung der Refactorings ausschließlich durch den Autor selbst durchgeführt wurde, kann eine subjektive Verzerrung nicht ausgeschlossen werden. Zur Minimierung dieses Risikos wurden die Bewertungskriterien vorab definiert und konsistent angewendet.

5.3.3. Vergleichbarkeit zwischen Projekten und Sprachen

Zwischen Sprachen und Projekten bestehen systematische Unterschiede. Python- und Apex-Projekte weichen trotz interner LOC-Angleichung in der Token-Anzahl für LLM-Inputs deutlich ab. Apex-Codebasen sind zudem doppelt so groß (LOC), was längere Prompts und Kontextprobleme verursacht, ein potenzieller Störfaktor für RQ2. Die LOC stellen dabei nur eine grobe Annäherung an die tatsächliche Token-Anzahl dar, die von den LLMs verarbeitet wird. Unterschiede in der Syntax oder String-Literalen können dazu führen, dass Projekte mit ähnlicher LOC dennoch unterschiedliche Tokenlängen im Prompt erzeugen. Dadurch kann die Vergleichbarkeit der Experimente zusätzlich beeinflusst werden. Zudem ist zu beachten, dass die LOC die Expressivität von Programmiersprachen nicht erfasst. Zwei Codebasen mit derselben Zeilenzahl können daher einen sehr unterschiedlichen Funktionsumfang besitzen. Dadurch kann auch der von LLMs zu verarbeitende Kontext unterschiedlich einfach oder schwierig sein, obwohl die Projekte anhand der LOC ähnlich groß wirken. Dieser Kritikpunkt wird teilweise dadurch abgefedert, dass in der vorliegenden Arbeit nicht ausschließlich absolute LOC betrachtet wurden, sondern mit $\frac{COC}{LOC}$ eine Verhältniskennzahl verwendet

wurde, die die Komplexität relativ zur Projektgröße beschreibt. Dadurch lässt sich die Dichte von Komplexität pro Zeile eines Projekts differenzierter erfassen als durch die reine LOC allein. Dennoch bleibt zu berücksichtigen, dass LOC auch in dieser Kennzahl im Nenner stehen und sprachübergreifend keine semantisch äquivalente Größe darstellen. Unterschiede in der Expressivität von Programmiersprachen können daher auch die Vergleichbarkeit von $\frac{COC}{LOC}$ zwischen Python und Apex einschränken. Die Refactorings selbst sind sprachenabhängig implementiert (*Getter-Setter* via `@property` in Python vs. explizite Methoden in Apex), was die Vergleichbarkeit erschwert. Lokale Python-Tests vs. remote Apex-Deployments führen zu unterschiedlichen Fehlerquellen. Zudem könnte ein gutes Abschneiden von Apex teilweise daran liegen, dass LLMs aus Trainingsdaten mehr Java-ähnliche und typische objektorientierte Muster kennen als syntaktisch unüblichere Sprachen [35]. Somit ist Apex nicht unbedingt ein besonders starker Testfall für eine weniger verbreitete Sprache, in der es potenziell weniger Trainingsdaten gibt. Die Ergebnisse zu RQ2 sollten deshalb vorsichtig interpretiert werden. Diese könnten eher einen Vergleich zwischen Python und einer Java-nahen Unternehmenssprache als zwischen Python und einer weniger verbreiteten Sprache mit weniger Trainingsdaten widerspiegeln.

5.3.4. Nicht deterministische Effekte und Methodik

Die nicht deterministische Natur der LLMs wurde durch zehn Iterationen pro Setup abgefedert, doch Ergebnisse können z.B. durch Serverlast verzerrt werden [58, 94]. Der systematische Prompt-Engineering-Prozess mit Few-Shot-Beispielen könnte einzelne Modelle begünstigen. Die Komplexitätsstufen (niedrig/mittel/hoch) sind zudem grob und nicht feingranuliert, ein mittleres Apex-Projekt könnte auch ein komplexes Python-Projekt sein. Aus den Ergebnissen lassen sich somit nur Trends ableiten, aber keine sicheren Beweise. Ein weiterer Aspekt betrifft die Interpretation der Projektkomplexität. Die Komplexität eines Projekts wird nicht ausschließlich durch einzelne schwierige Codeabschnitte bestimmt. Auch größere oder komplexere Projekte können Methoden enthalten, die isoliert sind und sich daher vergleichsweise leicht refactoren lassen. Da die Refactorings in dieser Arbeit überwiegend auf Methodenebene durchgeführt wurden, muss eine höhere Gesamtkomplexität eines Projekts nicht zwangsläufig zu schwierigeren Refactoring-Aufgaben führen.

5.3.5. Begrenzte Übertragbarkeit

Nur sechs spezifische Refactorings wurden getestet, andere Katalogeinträge (z. B. Extract Class, Pull Up Method) könnten abweichende Ergebnisse zeigen. Darüber hinaus ist nicht auszuschließen, dass andere Softwareprojekte deutlich größere Unterschiede in Struktur, Architektur oder Codequalität aufweisen. Projekte mit sehr hoher Komplexität oder stark gekoppelten Komponenten könnten daher andere Ergebnisse liefern als die in dieser Arbeit untersuchten Codebasen. Auch könnte eine Auswahl anderer LLMs zu anderen Ergebnissen führen.

5.4. Praktische Implikationen

Über die methodische Einordnung hinaus lassen sich aus den Ergebnissen auch praktische Schlussfolgerungen für den Einsatz von LLMs im Refactoring ableiten. Im Folgenden wird daher diskutiert, in welchen Szenarien die untersuchten Modelle sinnvoll eingesetzt werden können und wo weiterhin Grenzen bestehen.

5.4.1. Einsatzmöglichkeiten von LLM-basierten Refactorings

Die Ergebnisse dieser Arbeit deuten darauf hin, dass LLMs insbesondere bei kleineren und lokal begrenzten Refactorings zuverlässig eingesetzt werden können. Refactorings wie *Rename* oder *Inline Variable* konnten in vielen Fällen korrekt umgesetzt werden. In der Praxis könnten LLMs daher als unterstützende Werkzeuge für Entwickler dienen, um repetitive Refactoring-Aufgaben zu automatisieren.

5.4.2. Notwendigkeit menschlicher Kontrolle

Gleichzeitig zeigen die Ergebnisse, dass eine vollständig autonome Durchführung von Refactorings durch LLMs derzeit noch problematisch sein kann. Fehlerhafte Implementierungen oder unvollständige Änderungen traten weiterhin auf. Daher erscheint ein Einsatz als unterstützendes Werkzeug im Entwicklungsprozess, bei dem die erzeugten Änderungen von Entwicklern überprüft werden, sinnvoller.

5.4.3. Bedeutung einer ausreichenden Testabdeckung

Die Untersuchung unterstreicht zudem die Bedeutung einer umfangreichen Testabdeckung. Da die funktionale Korrektheit der Refactorings primär über bestehende Tests überprüft wurde, zeigt sich, dass LLM-basierte Refactorings insbesondere in Projekten mit gut ausgebauten Testfällen sinnvoll eingesetzt werden können.

5.4.4. Einfluss von Refactoring-Typ und Programmiersprache

Die Ergebnisse zeigen außerdem, dass die Erfolgsraten zwischen verschiedenen Refactoring-Typen variieren. Während einige Refactorings relativ zuverlässig umgesetzt wurden, erwiesen sich andere als deutlich schwieriger. In der Praxis sollten LLMs daher bevorzugt für klar definierte und strukturell einfache Refactorings eingesetzt werden. Darüber hinaus legen die Ergebnisse nahe, dass die Leistungsfähigkeit von LLMs beim Refactoring auch von der verwendeten Programmiersprache abhängen kann.

5.4.5. Unterschiede zur realen Softwareentwicklung

Es sollte außerdem berücksichtigt werden, dass die in dieser Arbeit verwendete Vorgehensweise nicht vollständig der typischen Nutzung von LLMs in der Softwareentwicklung entspricht. In der Praxis werden häufig nur relevante Codeausschnitte an ein

LLM übergeben, während in dieser Untersuchung der gesamte Projektcode als Kontext verwendet wurde. Darüber hinaus werden bestimmte Refactorings wie *Rename* häufig durch integrierte Entwicklungsumgebungen automatisiert durchgeführt. Moderne Werkzeuge wie *GitHub Copilot* [43] oder *Claude Code* [6] sind in der Regel direkt in Entwicklungsumgebungen integriert und werden interaktiv von Entwicklern genutzt. Entwickler können dabei gezielt einzelne Funktionen oder Codeabschnitte auswählen, zusätzliche Kontextinformationen bereitstellen und generierte Änderungen unmittelbar überprüfen oder anpassen [44]. Im Gegensatz dazu wurde in dieser Arbeit ein stärker kontrollierter experimenteller Ansatz gewählt. Den LLMs wurde jeweils der vollständige Projektcode sowie eine klar definierte Refactoring-Aufgabe übergeben. Die Modelle mussten daraufhin autonom eine Änderung am Projekt vornehmen, ohne dass eine iterative Interaktion mit dem Entwickler stattfand. In realen Entwicklungsprozessen werden LLMs hingegen häufig schrittweise verwendet, beispielsweise um einzelne Methoden umzuschreiben, Vorschläge zu generieren oder alternative Implementierungen zu diskutieren. Diese Unterschiede ziehen verschiedene Konsequenzen für die Interpretation der Ergebnisse nach sich. Einerseits könnte die Erfolgsrate in der Praxis höher sein, da Entwickler zusätzliche Kontextinformationen bereitstellen und generierte Änderungen schrittweise korrigieren können. Zum anderen ist es für integrierte Werkzeuge von Vorteil, wenn sie auf weiteren Kontext aus der Entwicklungsumgebung zugreifen können, beispielsweise über Symboltabellen, Projektnavigation oder Refactoring-Tools der IDE. So können gewisse Umwandlungen, wie beispielsweise das Ändern von Methodennamen, schon mit herkömmlichen Entwicklungswerkzeugen durchgeführt werden, während LLMs vor allem bei komplizierteren strukturellen Modifikationen als Hilfssystem dienen. Daher sollten die Resultate dieser Untersuchung in erster Linie als Beurteilung der Eignung von LLMs interpretiert werden, Refactorings auf Projektebene selbstständig vorzunehmen. In der Praxis werden LLMs meist in einem kollaborativen Kontext verwendet, als Hilfsmittel im Rahmen eines Entwicklungsprozesses.

5.4.6. Manueller Validierungsaufwand

Ein weiterer wichtiger Aspekt betrifft den praktischen Aufwand der Durchführung und Auswertung des Experiments. Insgesamt wurden in dieser Arbeit 720 Refactoring-Durchläufe durchgeführt. Jeder dieser Durchläufe musste anschließend manuell überprüft werden, um festzustellen, ob das jeweilige Refactoring korrekt umgesetzt wurde und ob die funktionale Korrektheit weiterhin gegeben war. Insbesondere die Bewertung der strukturellen Refactorings erforderte dabei eine sorgfältige Analyse des erzeugten Codes. Die manuelle Validierung dieser großen Anzahl an Iterationen stellte einen erheblichen zeitlichen Aufwand dar. Dadurch war es nur begrenzt möglich, die Anzahl der untersuchten Projekte, Refactoring-Typen oder Modelle weiter zu erhöhen. Eine größere Stichprobe hätte zwar potenziell zu noch robusteren Ergebnissen führen können, hätte jedoch den Rahmen dieser Arbeit deutlich überschritten. Der notwendige manuelle Bewertungsprozess stellt daher eine praktische Einschränkung der Arbeit dar und trägt zur begrenzten Generalisierbarkeit der Ergebnisse bei. Gleichzeitig zeigt dieser Aufwand, dass die Ergebnisse nicht ausschließlich auf automatisierten Metriken basieren, sondern durch eine detaillierte manuelle Analyse ergänzt wurden. Dies erhöht die

inhaltliche Aussagekraft der Bewertung, auch wenn dadurch die Größe des untersuchten Datensatzes eingeschränkt blieb.

5.5. Unerwartete Befunde

Im Rahmen der Durchführung trat eine Vielzahl unerwarteter Befunde auf. Entgegen der ursprünglichen Erwartung führte eine höhere Projektkomplexität nicht durchgängig zu geringeren Erfolgsraten. In einzelnen Fällen wurden bei komplexeren Projekten sogar bessere Ergebnisse beobachtet (vgl. Tabelle 4.1). Ebenfalls auffällig war, dass insbesondere *Guard Clauses* und *COC-Reduktion* überdurchschnittlich häufig fehlschlugen. Eine mögliche Erklärung ist, dass diese Aufgaben im Gegensatz zu anderen Refactorings keinen klar abgegrenzten Zielbereich vorgaben, sondern zunächst eine stärkere eigenständige Identifikation geeigneter Codeabschnitte erforderten.

6. Fazit und Ausblick

Dieses Kapitel widmet sich der Zusammenfassung der wichtigsten Ergebnisse sowie der Beantwortung der Forschungsfragen. Abschließend wird ein Ausblick auf mögliche zukünftige Forschungsansätze gegeben.

6.1. Zusammenfassung der wichtigsten Ergebnisse

Gemini 3 Pro stellt sich als ein sehr gutes Modell heraus, während *Codestral-2501* eher eine schlechte Performance zeigte. Die Erfolgsrate hängt zudem am stärksten von der Refactoring-Aufgabe ab. Zudem konnte diese Arbeit keinen Zusammenhang zwischen der Komplexität des Projekts und der Erfolgsrate aufzeigen. Jedoch lässt sich ein leichter Trend zwischen den Programmiersprachen erkennen. Doch dieser ist nur auf die Anzahl der untersuchten Projekte zurückzuführen und ermöglicht keine klare allgemeine Aussage, da das Experiment sehr eingeschränkt war. Abschließend ist hervorzuheben, dass die Aussagekraft der Ergebnisse durch mehrere Einschränkungen begrenzt ist. Insbesondere die relativ kleine Stichprobe an Projekten, Unterschiede in der Testabdeckung sowie die spezifische experimentelle Methodik beeinflussen die Generalisierbarkeit der Ergebnisse. Die beobachteten Resultate sollten daher primär als Hinweise auf mögliche Trends interpretiert werden und nicht als endgültige Aussagen über die Leistungsfähigkeit von LLMs beim automatisierten Refactoring.

6.2. Beantwortung der Forschungsfragen

6.2.1. RQ1: Inwieweit beeinflusst die Cognitive Complexity die Codequalität von LLM- Refactorings?

Die Ergebnisse dieser Arbeit zeigen, dass unter den Bedingungen des durchgeführten Experiments kein klarer Zusammenhang zwischen der COC eines Projekts und der Erfolgsrate von durch LLMs erzeugten Refactorings festgestellt werden konnte. Die Analyse der Erfolgsraten über die drei untersuchten Komplexitätsstufen, gering, mittel und komplex, zeigt, dass sich die durchschnittlichen Erfolgsraten nur geringfügig unterscheiden und kein konsistenter Trend erkennbar ist. Sowohl in Projekten mit niedriger als auch mit höherer Komplexität konnten vergleichbare Erfolgsraten beobachtet werden. Dieses Ergebnis deutet darauf hin, dass die Gesamtkomplexität eines Projekts nicht zwangsläufig einen starken Einfluss auf die Fähigkeit von LLMs hat, Refactorings korrekt umzusetzen. Eine mögliche Erklärung hierfür liegt darin, dass die Refactorings in dieser Arbeit überwiegend auf Methodenebene durchgeführt wurden. Selbst in größeren

oder komplexeren Codebasen können einzelne Methoden relativ isoliert sein und sich daher ähnlich leicht refactoren lassen wie in kleineren Projekten. Darüber hinaus zeigt die Analyse der Ergebnisse, dass andere Faktoren einen deutlich stärkeren Einfluss auf die Erfolgsrate haben als die Projektkomplexität. Insbesondere der Refactoring-Typ sowie das verwendete Modell haben sich als entscheidende Einflussfaktoren herausgestellt. Während einfache Refactorings häufig erfolgreich umgesetzt werden konnten, traten bei strukturell komplexeren Transformationen deutlich häufiger Fehler auf. Es ist jedoch zu beachten, dass die Aussagekraft dieser Ergebnisse durch mehrere Einschränkungen begrenzt ist. Die Anzahl der untersuchten Projekte ist relativ klein, und die Refactorings wurden nur auf ausgewählten Codeabschnitten durchgeführt. Daher lassen sich aus den Ergebnissen dieser Arbeit keine allgemeinen Aussagen darüber ableiten, ob Projektkomplexität grundsätzlich keinen Einfluss auf LLM-basierte Refactorings hat. Vielmehr deuten die Ergebnisse darauf hin, dass innerhalb der untersuchten Projekte und Refactorings kein signifikanter Zusammenhang beobachtet werden konnte.

6.2.2. RQ2: Sind die von LLMs erzeugten Refactorings bei weniger verbreiteten Sprachen wie Apex qualitativ ebenso hochwertig wie bei etablierten Sprachen wie Python?

Die Ergebnisse dieser Arbeit zeigen, dass sich zwischen den beiden untersuchten Programmiersprachen Unterschiede in den Erfolgsraten der durch LLMs erzeugten Refactorings erkennen lassen. Insgesamt wurden in Python-Projekten tendenziell höhere Erfolgsraten erzielt als in Apex-Projekten. Dieser Unterschied ist insbesondere bei einigen Refactoring-Typen sichtbar, bei denen die Modelle in Python-Projekten stabilere Ergebnisse liefern konnten. Gleichzeitig zeigen die Ergebnisse, dass dieser Unterschied stark vom verwendeten Modell abhängt. Während *Gemini 3 Pro* in beiden Programmiersprachen sehr hohe Erfolgsraten erreichte und nur geringe Unterschiede zwischen Apex und Python aufweist, zeigt *Codestral-2501* deutlich größere Unterschiede zwischen den beiden Sprachen. Dies deutet darauf hin, dass die Leistungsfähigkeit von LLMs beim automatisierten Refactoring nicht nur von der Programmiersprache selbst, sondern auch stark vom verwendeten Modell beeinflusst wird. Eine mögliche Erklärung für die beobachteten Unterschiede liegt darin, dass etablierte Programmiersprachen wie Python in vielen Trainingsdatensätzen deutlich stärker vertreten sind als spezialisiertere Sprachen wie Apex [35]. Dadurch verfügen LLMs möglicherweise über mehr Trainingsdaten und ein besseres Verständnis typischer Code-Strukturen und Idiome dieser Sprachen. Allerdings ist zu berücksichtigen, dass die Anzahl der untersuchten Projekte in dieser Arbeit begrenzt war und die Ergebnisse daher nur eingeschränkt generalisierbar sind. Die beobachteten Unterschiede zwischen Python und Apex könnten auch durch projektspezifische Eigenschaften, Unterschiede in der Testabdeckung oder sprachspezifische Implementierungsdetails einzelner Refactorings beeinflusst worden sein. Aus den vorliegenden Ergebnissen lässt sich daher kein allgemeingültiger Schluss ziehen, dass LLM-basierte Refactorings in weniger verbreiteten Programmiersprachen grundsätzlich schlechter funktionieren. Vielmehr zeigt die Untersuchung lediglich, dass innerhalb der betrachteten Projekte ein leichter Trend zugunsten von Python erkennbar ist.

6.3. Ausblick auf zukünftige Forschung

Die Ergebnisse dieser Arbeit zeigen mehrere Ansatzpunkte für zukünftige Forschung im Bereich LLM-basierter Refactorings. Ein zentraler Aspekt betrifft die Erweiterung der Datengrundlage. In dieser Arbeit wurden nur sechs Projekte untersucht, wodurch die Generalisierbarkeit der Ergebnisse eingeschränkt ist. Um statistisch signifikante Aussagen treffen zu können, wäre laut einer A-priori-Poweranalyse [36] bei einem mittleren Effekt (d = 0,335), $\alpha = 0,05$ und einer Power von 0,80 eine Stichprobengröße von n = 72 Projekten erforderlich gewesen. Die vorliegende Arbeit konnte diese Fallzahl wegen der zeitlichen Limitation nicht erreichen, was die statistische Aussagekraft einschränkt. Zukünftige Arbeiten könnten eine größere Anzahl an Projekten aus unterschiedlichen Anwendungsdomänen untersuchen, um robustere Aussagen über den Einfluss von Projektkomplexität, Programmiersprache und Refactoring-Typ zu ermöglichen. Darüber hinaus wäre es sinnvoll, weitere Refactoring-Arten zu analysieren. In dieser Arbeit wurden sechs ausgewählte Refactorings untersucht, die unterschiedliche Schwierigkeitsstufen abdecken. Es existieren jedoch eine Vielzahl weiterer Refactorings. Die Untersuchung solcher Refactorings könnte ein umfassenderes Bild darüber liefern, in welchen Bereichen LLMs besonders effektiv eingesetzt werden können. Ein weiterer wichtiger Forschungsansatz betrifft die Untersuchung zusätzlicher LLMs. Da sich in dieser Arbeit deutliche Leistungsunterschiede zwischen den beiden untersuchten Modellen gezeigt haben, könnte ein Vergleich mit weiteren aktuelleren Modellen zu neuen Erkenntnissen führen. Auch methodische Erweiterungen wären denkbar. Zukünftige Studien könnten beispielsweise automatisierte Bewertungsmethoden entwickeln, um größere Datensätze analysieren zu können und den manuellen Validierungsaufwand zu reduzieren. Dadurch ließen sich deutlich größere Experimente durchführen und statistisch belastbarere Aussagen treffen. Schließlich könnte zukünftige Forschung den praktischen Einsatz von LLMs in realen Entwicklungsumgebungen stärker untersuchen. Während in dieser Arbeit ein kontrolliertes experimentelles Setup verwendet wurde, arbeiten Entwickler in der Praxis häufig interaktiv mit LLM-basierten Werkzeugen. Studien, die diese interaktive Nutzung berücksichtigen, könnten zusätzliche Erkenntnisse über den tatsächlichen Nutzen von LLMs im Softwareentwicklungsprozess liefern.

Literaturverzeichnis

[1] *A New Era of Intelligence with Gemini 3*. Google. 18. Nov. 2025. URL: https://blog.google/products-and-platforms/products/gemini/gemini-3/ (besucht am 01. 03. 2026).

[2] Chaima Abid u. a. *30 Years of Software Refactoring Research:A Systematic Literature Review*. 4. Juli 2020. DOI: 10.48550/arXiv.2007.02194. arXiv: 2007.02194 [cs]. URL: http://arxiv.org/abs/2007.02194 (besucht am 26. 01. 2026). Vorveröffentlichung.

[3] *Aider LLM Leaderboards*. aider. URL: https://aider.chat/docs/leaderboards/ (besucht am 09. 03. 2026).

[6] *Anthropics/Claude-Code*. Anthropic, 9. März 2026.

[7] *Apex Rules | PMD Source Code Analyzer*. URL: https://pmd.github.io/pmd/pmd_rules_apex.html (besucht am 20. 02. 2026).

[14] *Better Code & Better Software | Ultimate Security and Quality*. URL: https://www.sonarsource.com/ (besucht am 20. 02. 2026).

[15] Rishi Bommasani u. a. *On the Opportunities and Risks of Foundation Models*. 12. Juli 2022. DOI: 10.48550/arXiv.2108.07258. arXiv: 2108.07258 [cs]. URL: http://arxiv.org/abs/2108.07258 (besucht am 16. 03. 2026). Vorveröffentlichung.

[18] Tom B. Brown u. a. *Language Models Are Few-Shot Learners*. 22. Juli 2020. DOI: 10.48550/arXiv.2005.14165. arXiv: 2005.14165 [cs]. URL: http://arxiv.org/abs/2005.14165 (besucht am 16. 03. 2026). Vorveröffentlichung.

[19] G. Ann Campbell. „Cognitive Complexity: An Overview and Evaluation“. In: *Proceedings of the 2018 International Conference on Technical Debt*. ICSE '18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, 27. Mai 2018, S. 57–58. DOI: 10.1145/3194164.3194186.

[20] Federico Cassano u. a. „MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation“. In: *IEEE Transactions on Software Engineering* 49.7 (Juli 2023), S. 3675–3691. DOI: 10.1109/TSE.2023.3267446.

[22] Mark Chen u. a. *Evaluating Large Language Models Trained on Code*. 14. Juli 2021. DOI: 10.48550/arXiv.2107.03374. arXiv: 2107.03374 [cs]. URL: http://arxiv.org/abs/2107.03374 (besucht am 27. 01. 2026). Vorveröffentlichung.

[24] Jinsu Choi, Gabin An und Shin Yoo. „Iterative Refactoring of Real-World Open-Source Programs with Large Language Models“. In: *Search-Based Software Engineering*. Hrsg. von Gunel Jahangirova und Foutse Khomh. Cham: Springer Nature Switzerland, 2024, S. 49–55. DOI: 10.1007/978-3-031-64573-0_4.

[25] *Codestral | Mistral AI*. URL: https://mistral.ai/news/codestral/ (besucht am 16. 03. 2026).

[26] *Codestral 25.01 | Mistral AI*. URL: https://mistral.ai/news/codestral-2501 (besucht am 12. 02. 2026).

[28] Jonathan Cordeiro, Shayan Noei und Ying Zou. *An Empirical Study on the Code Refactoring Capability of Large Language Models*. 4. Nov. 2024. DOI: 10.48550/arXiv.2411.02320. arXiv: 2411.02320 [cs]. URL: http://arxiv.org/abs/2411.02320 (besucht am 29. 12. 2025). Vorveröffentlichung.

[29] Jonathan Cordeiro, Shayan Noei und Ying Zou. „LLM-Driven Code Refactoring: Opportunities and Limitations“. In: *2025 IEEE/ACM Second IDE Workshop (IDE)*. 2025 IEEE/ACM Second IDE Workshop (IDE). Mai 2025, S. 32–36. DOI: 10.1109/IDE66625.2025.00011.

[31] Krishno Dey u. a. *Better to Ask in English: Evaluation of Large Language Models on English, Low-resource and Cross-Lingual Settings*. 17. Okt. 2024. DOI: 10.48550/arXiv.2410.13153. arXiv: 2410.13153 [cs]. URL: http://arxiv.org/abs/2410.13153 (besucht am 01. 03. 2026). Vorveröffentlichung.

[32] Taisei Enomoto u. a. „A Fair Comparison without Translationese: English vs. Target-language Instructions for Multilingual LLMs“. In: *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*. NAACL-HLT 2025. Hrsg. von Luis Chiruzzo, Alan Ritter und Lu Wang. Albuquerque, New Mexico: Association for Computational Linguistics, Apr. 2025, S. 649–670. DOI: 10.18653/v1/2025.naacl-short.55.

[33] *Entwicklerleitfaden für Gemini 3 | Gemini API | Google AI for Developers*. URL: https://ai.google.dev/gemini-api/docs/gemini-3?hl=de (besucht am 26. 01. 2026).

[34] OPDYKE W. F. „Refactoring : An Aid in Designing Application Frameworks and Evolving Object-oriented Systems“. In: *Proc. of 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)* (1990).

[35] Angela Fan u. a. *Large Language Models for Software Engineering: Survey and Open Problems*. 11. Nov. 2023. DOI: 10.48550/arXiv.2310.03533. arXiv: 2310.03533 [cs]. URL: http://arxiv.org/abs/2310.03533 (besucht am 16. 03. 2026). Vorveröffentlichung.

[36] Franz Faul u. a. „G*Power 3: A Flexible Statistical Power Analysis Program for the Social, Behavioral, and Biomedical Sciences“. In: *Behavior Research Methods* 39.2 (1. Mai 2007), S. 175–191. DOI: 10.3758/BF03193146.

[37] Norman Fenton und James Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, 1. Okt. 2014. 602 S. Google Books: lx_OBQAAQBAJ.

[38] Isabella Ferreira u. a. „Assessing the Bug-Proneness of Refactored Code: A Longitudinal Multi-Project Study“. In: *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. EASE '25. New York, NY, USA: Association for Computing Machinery, 24. Dez. 2025, S. 79–90. DOI: 10.1145/3756681.3756948.

[39] Martin Fowler und Kent Beck. *Refactoring: Improving the Design of Existing Code*. Second edition. The Addison-Wesley Signature Series. Boston Columbus New York San Francisco Amsterdam Cape Town Dubai London Munich: Addison-Wesley, 2019. 418 S.

[40] Martin Fowler, Kent Beck und John Brant. „Refactoring - Improving the Design of Existing Code“. In: ().

[41] *Gemini 3 Pro | Generative AI on Vertex AI*. Google Cloud Documentation. URL: https://docs.cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/3-pro?hl=de (besucht am 09. 03. 2026).

[42] *GitHub*. URL: https://github.com/ (besucht am 26. 01. 2026).

[43] *GitHub Copilot*. GitHub. URL: https://github.com/copilot (besucht am 06. 03. 2026).

[44] *GitHub Copilot Features*. GitHub Docs. URL: https://docs-internal.github.com/en/copilot/get-started/features?utm_source=chatgpt.com (besucht am 17. 03. 2026).

[46] Marwa Najm Abd Jader. „CALCULATING MCCABE'S CYCLOMATIC COMPLEXITY METRIC AND ITS EFFECT ON THE QUALITY ASPECTS OF SOFTWARE“. In: 3.5 ().

[48] Graylin Jay u. a. „Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship“. In: *Journal of Software Engineering and Applications* 02.03 (2009), S. 137–143. DOI: 10.4236/jsea.2009.23020.

[51] René Just. *Rjust/Defects4j*. 19. März 2026.

[52] Adam Tauman Kalai u. a. *Why Language Models Hallucinate*. 4. Sep. 2025. DOI: 10.48550/arXiv.2509.04664. arXiv: 2509.04664 [cs]. URL: http://arxiv.org/abs/2509.04664 (besucht am 27. 01. 2026). Vorveröffentlichung.

[53] Konstantin Kapitanov. „Apex Programming“. In: 1. Aug. 2024, S. 53–81. DOI: 10.1007/979-8-8688-0300-0_4.

[55] Philippe Kruchten, Robert L. Nord und Ipek Ozkaya. „Technical Debt: From Metaphor to Theory and Practice“. In: *IEEE Software* 29.6 (Nov. 2012), S. 18–21. DOI: 10.1109/MS.2012.167.

[56] Valentina Lenarduzzi, Terhi Kilamo und Andrea Janes. *Does Cyclomatic or Cognitive Complexity Better Represents Code Understandability? An Empirical Investigation on the Developers Perception*. Version 1. 14. März 2023. DOI: 10.48550/arXiv.2303.07722. arXiv: 2303.07722 [cs]. URL: http://arxiv.org/abs/2303.07722 (besucht am 27. 01. 2026). Vorveröffentlichung.

[57] Yaniv Leviathan, Matan Kalman und Yossi Matias. *Prompt Repetition Improves Non-Reasoning LLMs*. 17. Dez. 2025. DOI: 10.48550/arXiv.2512.14982. arXiv: 2512.14982 [cs]. URL: http://arxiv.org/abs/2512.14982 (besucht am 01. 03. 2026). Vorveröffentlichung.

[58] Fang Liu u. a. *Beyond Functional Correctness: Exploring Hallucinations in LLM-Generated Code*. 21. Jan. 2026. DOI: 10.48550/arXiv.2404.00971. arXiv: 2404.00971 [cs]. URL: http://arxiv.org/abs/2404.00971 (besucht am 26. 01. 2026). Vorveröffentlichung.

[59] Nelson F. Liu u. a. *Lost in the Middle: How Language Models Use Long Contexts*. 20. Nov. 2023. DOI: 10.48550/arXiv.2307.03172. arXiv: 2307.03172 [cs]. URL: http://arxiv.org/abs/2307.03172 (besucht am 16. 03. 2026). Vorveröffentlichung.

[61] Thainá Mariani und Silvia Regina Vergilio. „A Systematic Review on Search-Based Refactoring“. In: *Information and Software Technology* 83 (März 2017), S. 14–34. DOI: 10.1016/j.infsof.2016.11.009.

[62] Sofia Martinez u. a. „Software Refactoring Research with Large Language Models: A Systematic Literature Review“. In: *Journal of Systems and Software* 235 (1. Mai 2026), S. 112762. DOI: 10.1016/j.jss.2025.112762.

[63] T.J. McCabe. „A Complexity Measure“. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dez. 1976), S. 308–320. DOI: 10.1109/TSE.1976.233837.

[64] Shervin Minaee u. a. *Large Language Models: A Survey*. 23. März 2025. DOI: 10.48550/arXiv.2402.06196. arXiv: 2402.06196 [cs]. URL: http://arxiv.org/abs/2402.06196 (besucht am 26. 01. 2026). Vorveröffentlichung.

[66] Marvin Muñoz Barón, Marvin Wyrich und Stefan Wagner. „An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability“. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. Bari Italy: ACM, 5. Okt. 2020, S. 1–12. DOI: 10.1145/3382494.3410636.

[68] Ronald Nguyen. *Ronald-Nguyen/Bachelor-Thesis-Llm-Complexity*. 9. März 2026. URL: https://github.com/Ronald-Nguyen/bachelor-thesis-llm-complexity (besucht am 09. 03. 2026).

[69] Shiwen Ni u. a. *A Survey on Large Language Model Benchmarks*. 21. Aug. 2025. DOI: 10.48550/arXiv.2508.15361. arXiv: 2508.15361 [cs]. URL: http://arxiv.org/abs/2508.15361 (besucht am 16. 03. 2026). Vorveröffentlichung.

[70] *OpenAI Codex*. 13. März 2024. URL: https://openai.com/index/openai-codex/ (besucht am 16. 03. 2026).

[78] Simone Scalabrino u. a. „Automatically Assessing Code Understandability“. In: *IEEE Transactions on Software Engineering* 47.3 (März 2021), S. 595–613. DOI: 10.1109/TSE.2019.2901468.

[79] Sander Schulhoff u. a. *The Prompt Report: A Systematic Survey of Prompt Engineering Techniques*. 26. Feb. 2025. DOI: 10.48550/arXiv.2406.06608. arXiv: 2406.06608 [cs]. URL: http://arxiv.org/abs/2406.06608 (besucht am 09. 01. 2026). Vorveröffentlichung.

[81] Atsushi Shirafuji u. a. „Refactoring Programs Using Large Language Models with Few-Shot Examples". In: *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. 2023 30th Asia-Pacific Software Engineering Conference (APSEC). Dez. 2023, S. 151–160. DOI: 10.1109/APSEC60848.2023.00025.

[83] Marta Skreta u. a. *Errors Are Useful Prompts: Instruction Guided Task Programming with Verifier-Assisted Iterative Prompting*. 24. März 2023. DOI: 10.48550/arXiv.2303.14100. arXiv: 2303.14100 [cs]. URL: http://arxiv.org/abs/2303.14100 (besucht am 26. 02. 2026). Vorveröffentlichung.

[84] Saymon Souza u. a. *Beyond Strict Rules: Assessing the Effectiveness of Large Language Models for Code Smell Detection*. 14. Jan. 2026. DOI: 10.48550/arXiv.2601.09873. arXiv: 2601.09873 [cs]. URL: http://arxiv.org/abs/2601.09873 (besucht am 14. 03. 2026). Vorveröffentlichung.

[92] Ashish Vaswani u. a. „Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Bd. 30. Curran Associates, Inc., 2017.

[93] Dolores R Wallace, Arthur H Watson und Thomas J McCabe. *Structured Testing : A Testing Methodology Using the Cyclomatic Complexity Metric*. NIST SP 500-235. Gaithersburg, MD: National Institute of Standards and Technology, 1996, NIST SP 500–235. DOI: 10.6028/NIST.SP.500-235.

[94] Yanlin Wang u. a. *Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models*. 21. Juni 2025. DOI: 10.48550/arXiv.2407.00456. arXiv: 2407.00456 [cs]. URL: http://arxiv.org/abs/2407.00456 (besucht am 16. 03. 2026). Vorveröffentlichung.

[95] Zichong Wang u. a. *History, Development, and Principles of Large Language Models-An Introductory Survey*. 23. Sep. 2024. DOI: 10.48550/arXiv.2402.06853. arXiv: 2402.06853 [cs]. URL: http://arxiv.org/abs/2402.06853 (besucht am 26. 01. 2026). Vorveröffentlichung.

[96] *What Is Apex? | Apex Developer Guide | Salesforce Developers*. URL: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_intro_what_is_apex.htm (besucht am 07. 02. 2026).

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt und ohne fremde Hilfe verfasst habe. Dazu habe ich keine außer den von mir angegebenen Hilfsmitteln und Quellen verwendet und die den benutzten Werken inhaltlich und wörtlich entnommenen Stellen habe ich als solche kenntlich gemacht. Ich versichere, dass die eingereichte elektronische Fassung mit den gedruckten Exemplaren übereinstimmt.

Rostock, den 19.03.2026

Ronald Nguyen

A. Anhang

A.1. Vollständiger Prompt – Rename (Python)

```
Rename
Task Description
You are a strict code-refactoring engine. You follow all rules exactly as written.
Your task is to perform a refactoring change in a Python project.
A function needs to be renamed.
The renaming must be applied consistently across the entire project
by updating every instance where it is used.

Please note:
1. The code must still function perfectly after the renaming.
2. All imports, references, and function calls must be updated accordingly.
3. The style, formatting, and structure of the code must be preserved.
4. Do not introduce any new functionality, logic changes, or unrelated formatting changes.
5. For every modified file, return the complete updated file content to ensure that no changes are overlooked.
6. The semantics and behavior of the code must not change.
7. Only include files that required changes due to the renaming.
8. Respond with the code blocks only. Your entire response must be machine-readable. Do not include any conversational filler.
9. If you are unsure, do not guess. Only modify code when the change is certain.
10. Every response must include meaningful modifications, improvements, or transformations. Returning the same code, even partially or
↪  with superficial edits, is strictly prohibited.
11. Your final output must pass the existing test suite. If your changes would cause any test to fail, revise your refactor until all
↪  tests pass.

Examples of renaming tasks:
Example 1:
Original Code:
File `calculate.py`:
def calculate_sum(a, b):
    return a + b

print(calculate_sum(2, 3))

Task:
1. Rename the method `calculate_sum` in the file: `calculate.py` to `add`.
2. Update every relevant occurrence across the entire project.
3. Output format (for EACH modified file):

File `calculate.py`:
def add(a, b):
    return a + b

print(add(2, 3))

Example 2:
Original Code:
File `max.py`:
def find_maximum(numbers):
    return max(numbers)

result = find_maximum([1, 2, 3])

Task:
1. Rename the method `find_maximum` in the file: `max.py` to `get_max`.
2. Update every relevant occurrence across the entire project.
3. Output format (for EACH modified file):

File `max.py`:
def get_max(numbers):
    return max(numbers)

result = get_max([1, 2, 3])

Your Task:
1. Rename the method `should_wrap` in the file: `ansitowin32.py` to `is_wrappable`
2. Update every relevant occurrence across the entire project.
3. For EACH modified file, respond exactly in the following format:

File `filename.py`:
[Complete updated file content]
```

A.2. Übersicht aller experimentellen Ergebnisse

Sprache	Projekt	Komplexität	Refactoring	schwierigkeit	LLM	Diff	Test	Refactoring-check	Gesamt
Apex	apex-utilities	gering	coc reduktion	schwierig	codestral-2501	100%	10%	10%	10%
Apex	apex-utilities	gering	coc reduktion	schwierig	Gemini 3 Pro	100%	100%	100%	100%
Apex	force-di	mittel	coc reduktion	schwierig	codestral-2501	100%	30%	100%	30%
Apex	force-di	mittel	coc reduktion	schwierig	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-dml-mocking	komplex	coc reduktion	schwierig	codestral-2501	100%	0%	0%	0%
Apex	apex-dml-mocking	komplex	coc reduktion	schwierig	Gemini 3 Pro	100%	60%	50%	30%
Python	colorama	gering	coc reduktion	schwierig	codestral-2501	100%	40%	0%	0%
Python	colorama	gering	coc reduktion	schwierig	Gemini 3 Pro	100%	100%	100%	100%
Python	pluggy	mittel	coc reduktion	schwierig	codestral-2501	100%	60%	40%	30%
Python	pluggy	mittel	coc reduktion	schwierig	Gemini 3 Pro	100%	60%	80%	60%
Python	pathlib2	komplex	coc reduktion	schwierig	codestral-2501	20%	100%	0%	0%
Python	pathlib2	komplex	coc reduktion	schwierig	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-utilities	gering	getter setter	mittel	codestral-2501	100%	0%	50%	0%
Apex	apex-utilities	gering	getter setter	mittel	Gemini 3 Pro	100%	100%	100%	100%
Apex	force-di	mittel	getter setter	mittel	codestral-2501	100%	100%	100%	100%
Apex	force-di	mittel	getter setter	mittel	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-dml-mocking	komplex	getter setter	mittel	codestral-2501	100%	0%	100%	0%
Apex	apex-dml-mocking	komplex	getter setter	mittel	Gemini 3 Pro	100%	100%	100%	100%
Python	colorama	gering	getter setter	mittel	codestral-2501	100%	100%	100%	100%
Python	colorama	gering	getter setter	mittel	Gemini 3 Pro	100%	100%	100%	100%
Python	pluggy	mittel	getter setter	mittel	codestral-2501	100%	100%	100%	100%
Python	pluggy	mittel	getter setter	mittel	Gemini 3 Pro	100%	100%	100%	100%
Python	pathlib2	komplex	getter setter	mittel	codestral-2501	100%	100%	100%	100%
Python	pathlib2	komplex	getter setter	mittel	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-utilities	gering	guard clauses	mittel	codestral-2501	90%	100%	40%	40%
Apex	apex-utilities	gering	guard clauses	mittel	Gemini 3 Pro	100%	100%	90%	90%
Apex	force-di	mittel	guard clauses	mittel	codestral-2501	90%	100%	30%	30%
Apex	force-di	mittel	guard clauses	mittel	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-dml-mocking	komplex	guard clauses	mittel	codestral-2501	90%	70%	30%	10%
Apex	apex-dml-mocking	komplex	guard clauses	mittel	Gemini 3 Pro	90%	100%	50%	50%
Python	colorama	gering	guard clauses	mittel	codestral-2501	50%	100%	100%	50%
Python	colorama	gering	guard clauses	mittel	Gemini 3 Pro	100%	100%	100%	100%
Python	pluggy	mittel	guard clauses	mittel	codestral-2501	10%	100%	0%	0%
Python	pluggy	mittel	guard clauses	mittel	Gemini 3 Pro	100%	100%	100%	100%
Python	pathlib2	komplex	guard clauses	mittel	codestral-2501	100%	100%	100%	100%
Python	pathlib2	komplex	guard clauses	mittel	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-utilities	gering	inline variable	einfach	codestral-2501	100%	100%	80%	80%
Apex	apex-utilities	gering	inline variable	einfach	Gemini 3 Pro	100%	100%	100%	100%
Apex	force-di	mittel	inline variable	einfach	codestral-2501	100%	0%	50%	0%
Apex	force-di	mittel	inline variable	einfach	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-dml-mocking	komplex	inline variable	einfach	codestral-2501	100%	100%	90%	90%
Apex	apex-dml-mocking	komplex	inline variable	einfach	Gemini 3 Pro	100%	100%	100%	100%
Python	colorama	gering	inline variable	einfach	codestral-2501	100%	80%	100%	80%
Python	colorama	gering	inline variable	einfach	Gemini 3 Pro	100%	80%	100%	80%
Python	pluggy	mittel	inline variable	einfach	codestral-2501	80%	100%	80%	80%
Python	pluggy	mittel	inline variable	einfach	Gemini 3 Pro	100%	100%	100%	100%
Python	pathlib2	komplex	inline variable	einfach	codestral-2501	100%	100%	100%	100%
Python	pathlib2	komplex	inline variable	einfach	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-utilities	gering	rename	einfach	codestral-2501	100%	90%	80%	80%
Apex	apex-utilities	gering	rename	einfach	Gemini 3 Pro	100%	100%	100%	100%
Apex	force-di	mittel	rename	einfach	codestral-2501	100%	100%	100%	100%
Apex	force-di	mittel	rename	einfach	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-dml-mocking	komplex	rename	einfach	codestral-2501	100%	100%	100%	100%
Apex	apex-dml-mocking	komplex	rename	einfach	Gemini 3 Pro	100%	100%	100%	100%
Python	colorama	gering	rename	einfach	codestral-2501	100%	60%	100%	60%
Python	colorama	gering	rename	einfach	Gemini 3 Pro	100%	100%	100%	100%
Python	pluggy	mittel	rename	einfach	codestral-2501	100%	90%	100%	90%
Python	pluggy	mittel	rename	einfach	Gemini 3 Pro	100%	100%	100%	100%
Python	pathlib2	komplex	rename	einfach	codestral-2501	100%	80%	100%	80%
Python	pathlib2	komplex	rename	einfach	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-utilities	gering	strategy pattern	schwierig	codestral-2501	100%	10%	100%	10%
Apex	apex-utilities	gering	strategy pattern	schwierig	Gemini 3 Pro	100%	100%	100%	100%
Apex	force-di	mittel	strategy pattern	schwierig	codestral-2501	100%	0%	100%	0%
Apex	force-di	mittel	strategy pattern	schwierig	Gemini 3 Pro	100%	100%	100%	100%
Apex	apex-dml-mocking	komplex	strategy pattern	schwierig	codestral-2501	100%	100%	100%	100%
Apex	apex-dml-mocking	komplex	strategy pattern	schwierig	Gemini 3 Pro	100%	100%	100%	100%
Python	colorama	gering	strategy pattern	schwierig	codestral-2501	30%	100%	10%	10%
Python	colorama	gering	strategy pattern	schwierig	Gemini 3 Pro	100%	100%	100%	100%
Python	pluggy	mittel	strategy pattern	schwierig	codestral-2501	100%	10%	100%	10%
Python	pluggy	mittel	strategy pattern	schwierig	Gemini 3 Pro	100%	100%	100%	100%
Python	pathlib2	komplex	strategy pattern	schwierig	codestral-2501	100%	40%	100%	40%
Python	pathlib2	komplex	strategy pattern	schwierig	Gemini 3 Pro	100%	100%	100%	100%

Legende

Test Anteil bestandener Regressionstests.

Refactoring-check Anteil korrekt implementierter Refactorings.

Diff Anteil an Iterationen, in denen eine Änderung vorgenommen wurde.

Gesamt Anteil der Durchläufe, bei denen sowohl Tests bestanden als auch das Refactoring korrekt umgesetzt wurde.

A.3. Referenzen zu allen untersuchten Projekten

[4] Andi Albrecht. *Andialbrecht/Sqlparse*. 6. März 2026. URL: https://github.com/andialbrecht/sqlparse (besucht am 06. 03. 2026).

[5] *Amoffat/Sh: Python Process Launching*. URL: https://github.com/amoffat/sh (besucht am 06. 03. 2026).

[8] *Apex-Enterprise-Patterns/At4dx-Samplecode: Sample Code Demonstrating the Usage of the AT4DX Framework*. URL: https://github.com/apex-enterprise-patterns/at4dx-samplecode (besucht am 06. 03. 2026).

[9] *Apex-Enterprise-Patterns/Fflib-Apex-Common-Samplecode: Samples Application Illustrating the Apex Enterprise Patterns Library*. URL: https://github.com/apex-enterprise-patterns/fflib-apex-common-samplecode (besucht am 06. 03. 2026).

[10] *Apex-Enterprise-Patterns/Fflib-Apex-Common: Common Apex Library Supporting Apex Enterprise Patterns and Much More!* URL: https://github.com/apex-enterprise-patterns/fflib-apex-common (besucht am 06. 03. 2026).

[11] *Apex-Enterprise-Patterns/Fflib-Apex-Mocks: An Apex Mocking Framework for True Unit Testing in Salesforce, with Stub API Support*. URL: https://github.com/apex-enterprise-patterns/fflib-apex-mocks (besucht am 06. 03. 2026).

[12] *Apex-Enterprise-Patterns/Force-Di: Generic DI Library with Support for Apex, Triggers, Visualforce and Lightning*. URL: https://github.com/apex-enterprise-patterns/force-di (besucht am 06. 03. 2026).

[13] Anton B. *Ab77/Netflix-Proxy*. 5. März 2026. URL: https://github.com/ab77/netflix-proxy (besucht am 06. 03. 2026).

[16] *Bottlepy/Bottle*. Bottle Micro Web Framework. 6. März 2026. URL: https://github.com/bottlepy/bottle (besucht am 06. 03. 2026).

[17] *Brianmfear/Apex-Lambda: Functional Programming for Salesforce Apex*. URL: https://github.com/brianmfear/apex-lambda (besucht am 06. 03. 2026).

[21] *Certinia/Apex-Mdapi: Apex Wrapper for the Salesforce Metadata API*. URL: https://github.com/certinia/apex-mdapi (besucht am 06. 03. 2026).

[23] Benoit Chesneau. *Benoitc/Gunicorn*. 6. März 2026. URL: https://github.com/benoitc/gunicorn (besucht am 06. 03. 2026).

[27] Samuel Colvin u. a. *Pydantic Validation*. Version v2.13.0b2. Feb. 2026. URL: https://github.com/pydantic/pydantic (besucht am 06. 03. 2026).

[30] *Delgan/Loguru: Python Logging Made (Stupidly) Simple*. URL: https://github.com/Delgan/loguru (besucht am 06. 03. 2026).

[45] *Google/Yapf*. Google. 6. März 2026. URL: https://github.com/google/yapf (besucht am 06. 03. 2026).

[47] *Jamessimone/Apex-Rollup: Fast, Configurable, Elastically Scaling Custom Rollup Solution. Apex Invocable Action, One-Liner Apex Trigger/CMDT-driven Logic, and Scheduled Apex-ready.* URL: https://github.com/jamessimone/apex-rollup (besucht am 06. 03. 2026).

[49] *Jazzband/Pathlib2: Backport of Pathlib Aiming to Support the Full Stdlib Python API.* URL: https://github.com/jazzband/pathlib2 (besucht am 06. 03. 2026).

[50] *Jongpie/NebulaLogger: The Most Robust Observability Solution for Salesforce Experts. Built 100% Natively on the Platform, and Designed to Work Seamlessly with Apex, Lightning Components, Flow, OmniStudio, and Integrations.* URL: https://github.com/jongpie/NebulaLogger (besucht am 06. 03. 2026).

[54] *Kevinohara80/Sfdc-Trigger-Framework: A Minimal Trigger Framework for Your Salesforce Apex Triggers.* URL: https://github.com/kevinohara80/sfdc-trigger-framework (besucht am 06. 03. 2026).

[60] Rahul Malhotra. *Rahulmalhotra/HTTPCalloutFramework.* 18. Jan. 2026. URL: https://github.com/rahulmalhotra/HTTPCalloutFramework (besucht am 06. 03. 2026).

[65] *Msiemens/Tinydb: TinyDB Is a Lightweight Document Oriented Database Optimized for Your Happiness :)* URL: https://github.com/msiemens/tinydb (besucht am 06. 03. 2026).

[67] Val Neekman. *Un33k/Python-Slugify.* 5. März 2026. URL: https://github.com/un33k/python-slugify (besucht am 06. 03. 2026).

[71] *Pallets/Jinja.* Pallets. 6. März 2026. URL: https://github.com/pallets/jinja (besucht am 06. 03. 2026).

[72] Gavin Palmer. *Gavinhughpalmer/Apex-Utilities.* 23. Sep. 2024. URL: https://github.com/gavinhughpalmer/apex-utilities (besucht am 06. 03. 2026).

[73] *Psf/Requests.* Python Software Foundation. 6. März 2026. URL: https://github.com/psf/requests (besucht am 06. 03. 2026).

[74] *Pytest-Dev/Pluggy: A Minimalist Production Ready Plugin System.* URL: https://github.com/pytest-dev/pluggy (besucht am 06. 03. 2026).

[75] *Python-Markdown/Markdown.* Python-Markdown. 6. März 2026. URL: https://github.com/Python-Markdown/markdown (besucht am 06. 03. 2026).

[76] *Rsoesemann/Apex-Domainbuilder: Framework to Setup Apex Test Data in a Highly Flexible and Readable Way Using the Test Data Builder Pattern.* URL: https://github.com/rsoesemann/apex-domainbuilder (besucht am 06. 03. 2026).

[77] Robert SÃ¶semann. *Rsoesemann/Apex-Unified-Logging.* 5. März 2026. URL: https://github.com/rsoesemann/apex-unified-logging (besucht am 06. 03. 2026).

[80] *Sendgrid/Sendgrid-Apex.* Twilio SendGrid. 11. Feb. 2026. URL: https://github.com/sendgrid/sendgrid-apex (besucht am 06. 03. 2026).

[82] James Simone. *Jamessimone/Apex-Dml-Mocking.* 27. Feb. 2026. URL: https://github.com/jamessimone/apex-dml-mocking (besucht am 06. 03. 2026).

[85] Mitch Spano. *Mitchspano/Trigger-Actions-Framework*. 5. März 2026. URL: https://github.com/mitchspano/trigger-actions-framework (besucht am 06. 03. 2026).

[86] *Tartley/Colorama: Simple Cross-Platform Colored Terminal Text in Python*. URL: https://github.com/tartley/colorama (besucht am 06. 03. 2026).

[87] *Tornadoweb/Tornado: Tornado Is a Python Web Framework and Asynchronous Networking Library, Originally Developed at FriendFeed*. URL: https://github.com/tornadoweb/tornado (besucht am 06. 03. 2026).

[88] *Trailheadapps/Apex-Recipes*. Salesforce Developers Sample Apps. 6. März 2026. URL: https://github.com/trailheadapps/apex-recipes (besucht am 06. 03. 2026).

[89] *Trailheadapps/Automation-Components*. Salesforce Developers Sample Apps. 28. Feb. 2026. URL: https://github.com/trailheadapps/automation-components (besucht am 06. 03. 2026).

[90] *Trailheadapps/Dreamhouse-Lwc: Sample Application for Lightning Web Components on Salesforce Platform. Part of the Sample Gallery. Real Estate Use Case. Get Inspired and Learn Best Practices*. URL: https://github.com/trailheadapps/dreamhouse-lwc (besucht am 06. 03. 2026).

[91] *UnofficialSF/LightningFlowComponents: A Collection of Unofficial Flow Extensions That Can Be Used to Enhance Salesforce Flow and Orchestrator*. URL: https://github.com/UnofficialSF/LightningFlowComponents (besucht am 06. 03. 2026).

[97] *Yaml/Pyyaml*. The YAML Project. 6. März 2026. URL: https://github.com/yaml/pyyaml (besucht am 06. 03. 2026).

A.4. Relevanter Codeabschnitt im Guard Clauses Refactoring nach der Transformation

Listing A.1.: Transformierter Abschnitt der Methode `getRepoFromSObjectType`

```
if (alwaysUseMock) {
  RepoMock mock = new RepoMock(sObjectType, repoFactory);
  List<AggregateRecord> aggRecords = AggregateResults.get(sObjectType);
  if (aggRecords != null) {
    mock.aggRecords.addAll(aggRecords);
  }
  List<FieldLevelHistory> historyRecords = HistoryResults.get(sObjectType);
  if (historyRecords != null) {
    mock.historyRecords.addAll(historyRecords);
  }
  return mock;
}

List<SObject> queriedResults = getResults(sObjectType);
if (!queriedResults.isEmpty()) {
  RepoMock mock = new RepoMock(sObjectType, repoFactory);
  List<AggregateRecord> aggRecords = AggregateResults.get(sObjectType);
  if (aggRecords != null) {
    mock.aggRecords.addAll(aggRecords);
  }

  List<FieldLevelHistory> historyRecords = HistoryResults.get(sObjectType);
  if (historyRecords != null) {
    mock.historyRecords.addAll(historyRecords);
  }

  List<Cursor> cursorResults = CursorResults.get(sObjectType);
  if (cursorResults != null && !cursorResults.isEmpty()) {
    return mock;
  }
}
```

A.5. Relevanter Codeabschnitt im Strategy Refactoring nach der Transformation

Listing A.2.: Transformierter Abschnitt der Methode `call_win32`

```
...
def call_win32(self, command, params):
    if command == 'm':
        for param in params:
            if param in self.win32_calls:
                func_args = self.win32_calls[param]
                func = func_args[0]
                args = func_args[1:]
                kwargs = dict(on_stderr=self.on_stderr)
                func(*args, **kwargs)
    elif command in 'J':
        winterm.erase_screen(params[0], on_stderr=self.on_stderr)
    elif command in 'K':
        winterm.erase_line(params[0], on_stderr=self.on_stderr)
    elif command in 'Hf':
        winterm.set_cursor_position(params, on_stderr=self.on_stderr)
    elif command in 'ABCD':
        n = params[0]
        x, y = {'A': (0, -n), 'B': (0, n), 'C': (n, 0), 'D': (-n, 0)}[command]
        winterm.cursor_adjust(x, y, on_stderr=self.on_stderr)

def _strategy_m(self, params):
    for param in params:
        if param in self.win32_calls:
            func_args = self.win32_calls[param]
            func = func_args[0]
            args = func_args[1:]
            kwargs = dict(on_stderr=self.on_stderr)
            func(*args, **kwargs)

def _strategy_J(self, params):
    winterm.erase_screen(params[0], on_stderr=self.on_stderr)

def _strategy_K(self, params):
    winterm.erase_line(params[0], on_stderr=self.on_stderr)

def _strategy_Hf(self, params):
    winterm.set_cursor_position(params, on_stderr=self.on_stderr)

def _strategy_ABCD(self, params):
    n = params[0]
    x, y = {'A': (0, -n), 'B': (0, n), 'C': (n, 0), 'D': (-n, 0)}[command]
    winterm.cursor_adjust(x, y, on_stderr=self.on_stderr)

WIN32_STRATEGIES = {
    'm': _strategy_m,
    'J': _strategy_J,
    'K': _strategy_K,
    'H': _strategy_Hf,
    'f': _strategy_Hf,
    'A': _strategy_ABCD,
    'B': _strategy_ABCD,
    'C': _strategy_ABCD,
    'D': _strategy_ABCD,
}

def call_win32(self, command, params):
    if command in self.WIN32_STRATEGIES:
        self.WIN32_STRATEGIES[command](params)
...
```

A.6. Relevanter Ausschnitt der erzeugten Änderung bei der COC-Reduktion

Listing A.3.: Gekürzte Version der Veränderung im Refactoring: *COC-Reduktion*

```
...
--- backup/repository\Repository.cls
+++ refactored/repository\Repository.cls
@@ -20,0 +21,3 @@
+if(repoType==null||queryFields==null||repoFactory==null){
+thrownewIllegalArgumentException('Repositoryparameterscannotbenull');
+}
@@ -26,0 +30,3 @@
+if(queries==null){
+queries=newList<Query>();
+}
@@ -34,0 +41,3 @@
+if(queries==null){
+queries=newList<Query>();
+}
@@ -37,0 +47,3 @@
+if(queries==null){
+queries=newList<Query>();
+}
@@ -51,0 +64,3 @@
+if(query==null){
+returnthis.get(newList<Query>());
+}
@@ -54,0 +70,3 @@
+if(queries==null){
+queries=newList<Query>();
+}
@@ -65,0 +84,3 @@
+if(fieldToken==null||sortOrder==null){
+returnthis;
+}
@@ -69,0 +91,3 @@
+if(parentFieldChain==null||parentFieldChain.isEmpty()||sortOrder==null){
+returnthis;
+}
@@ -73,0 +98,3 @@
+if(fields==null||fields.isEmpty()){
+returnthis;
+}
...
```