

MASTERARBEIT

zur Erlangung des akademischen Grades
„Master of Science in Engineering“
im Studiengang Maschinenbau

OptiPy: Automatisierte Optimierung der inneren Softwarequalität von Python-Code mithilfe von Clean-Code-Prinzipien und einem *Large Language-Modell*

Ausgeführt von: Philipp Gasser, BSc
Personenkennzeichen: 2310804002

1. BegutachterIn: FH-Prof. PD DI Dr. techn. Maximilian Lackner, MBA
2. BegutachterIn: Mag. Dr. Markus Pak

Wien, 14.05.2025

Eidesstattliche Erklärung

„Ich, als Autor / als Autorin und Urheber / Urheberin der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. Urheberrechtsgesetz idGf sowie Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idGf).

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nach den aktuell geltenden Regeln der FH Technikum Wien angefertigt und dass ich Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idGf).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Ort, Datum

Digitale Unterschrift

Kurzfassung

Im letzten Jahrzehnt hat sich Python dank seiner einfachen Syntax, der kostenfreien Nutzung und der Vielzahl an Bibliotheken als eine der beliebtesten Programmiersprachen etabliert. Diese Eigenschaften machen Python sowohl für Einsteiger (laut PYPL-Index) als auch für erfahrene Entwickler (laut TIOBE-Index) attraktiv.

Die Kombination verschiedener Erfahrungsgrade innerhalb eines Teams kann zu erheblichen Unterschieden in der Codequalität führen. Dieser Effekt wird durch die zunehmende Popularität von Coding-Assistenten, wie ChatGPT oder GitHub Copilot, verstärkt. Eine hohe Codequalität ist jedoch essenziell, um eine effiziente und effektive Zusammenarbeit zu gewährleisten. Aus Entwicklerperspektive sind insbesondere innere Qualitätsmerkmale, wie Wartbarkeit, Zuverlässigkeit und Performance-Effizienz relevant.

Im Rahmen dieser Arbeit wird daher ein Clean-Code-Leitfaden für die Python-Entwicklung entworfen, der zur Verbesserung der inneren Softwarequalität beiträgt. Der Leitfaden basiert auf anerkannten Python-Enhancement-Proposals (PEP-8, PEP-20, PEP-257, PEP-282, PEP-287 und PEP-484) und auf *Best Practices* der Fachliteratur.

Anschließend wird eine Anwendung namens OptiPy entwickelt, die den Leitfaden mithilfe eines Large Language Modells automatisiert auf eine gegebene Python-Datei anwendet. OptiPy kann dabei nahtlos in bestehende Python-Anwendungen importiert, über die Kommandozeile genutzt oder direkt in CI/CD-Pipelines integriert werden. Zur Evaluierung der Praktikabilität werden die Flagship-Modelle von Anthropic (Claude 3.7 Sonnet), Google (Gemini 2.0 Pro), OpenAI (GPT-4o) und xAI (Grok 2) anhand von drei Fallstudien und in insgesamt 120 Durchläufen untersucht. Diese Szenarien differenzieren sich hinsichtlich der Entwicklungserfahrung sowie der damit einhergehenden Anzahl an *Code-Smells*.

Die Ergebnisse der Fallstudien zeigen, dass die iterative Anwendung der einzelnen Kapitel des Clean-Code-Leitfadens („One-by-One“) zwar kosten- und zeitintensiver ist als eine ganzheitliche Überarbeitung des Codes („All-at-Once“), jedoch zu einer geringeren Anzahl an Regelverstößen führt. Besonders hervorzuheben ist GPT-4o, welches in keiner der Untersuchungen Regelverstöße verzeichnete. Weiters ist es insbesondere für professionelle Entwickler:innen empfehlenswert, den Leitfaden bedarfsgerecht zu komprimieren, um den Optimierungsprozess hinsichtlich Kosten und Dauer effizienter zu gestalten.

Ein Wirtschaftlichkeitsvergleich auf Basis eines monatlichen Arbeitnehmerbruttogehalts von 4500€ zeigt, dass bereits marginale Zeitersparnisse von 3,8% bis 8,0% (je nach Unternehmensgröße) in Code-Reviews ausreichen, um den Einsatz von OptiPy wirtschaftlich zu rechtfertigen. Unter der konservativen Annahme, dass 33% der Zeit in Code-Reviews eingespart werden können, beträgt die jährliche Ersparnis des Arbeitgebers pro Entwickler:in 2065€ bis 2335€.

Schlagwörter: Python, Clean Code, Innere Softwarequalität, Wartbarkeit, LLM, API

Abstract

In the last decade, Python has established itself as one of the most popular programming languages thanks to its simple syntax, free availability, and the wide range of libraries. These characteristics make Python attractive for both beginners (according to the PYPL index) and experienced developers (according to the TIOBE index).

Varying experience levels within a team can result in significant disparities in code quality. This effect is amplified by the increasing popularity of coding assistants such as ChatGPT or GitHub Copilot. However, high code quality is essential to ensure efficient and effective collaboration. From a developer's perspective, internal quality features such as maintainability, reliability and performance efficiency are particularly relevant.

This thesis therefore develops a clean code guideline for Python development that contributes to improving internal software quality. The guideline is based on recognised Python enhancement proposals (PEP-8, PEP-20, PEP-257, PEP-282, PEP-287 and PEP-484) and on established best practices from the literature.

An application called OptiPy is then developed, which automatically applies the guideline to a given Python file using a large language model. OptiPy can be seamlessly imported into existing Python applications, used via the command line or integrated directly into CI/CD pipelines. To evaluate practicability, the flagship models from Anthropic (Claude 3.7 Sonnet), Google (Gemini 2.0 Pro), OpenAI (GPT-4o) and xAI (Grok 2) are analysed using three case studies and 120 runs. These scenarios differ in terms of development experience and the associated number of code smells.

The results of the case studies show that the iterative application of the individual chapters of the clean code guideline („One-by-One“) is more costly and time-consuming than a holistic revision of the code („All-at-Once“), but results in fewer rule violations. GPT-4o is particularly noteworthy, as it did not exhibit any rule violations in any of the analyses. Furthermore, it is particularly advisable for professional developers to tailor the guidelines as needed in order to make the optimisation process more efficient in terms of costs and duration.

A profitability comparison based on a gross monthly employee salary of €4500 shows that even marginal time savings of 3.8% to 8.0% (depending on company size) in code reviews are sufficient to economically justify the use of OptiPy. Under the conservative assumption that 33% of the time in code reviews can be saved, the annual savings for the employer per developer amount to €2065 to €2335.

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben.

Zuerst gebührt mein Dank Herrn PD DI Dr. Maximilian Lackner, MBA und Herrn Mag. Dr. Markus Pak, die meine Masterarbeit betreut und begutachtet haben. Ich möchte mich herzlich für die regelmäßigen, spannenden und inspirierenden Jour-Fixes bedanken. Für die hilfreichen Anregungen, die schnelle Abrufbarkeit und die konstruktive Kritik bei der Erstellung dieser Arbeit bin ich sehr dankbar. Ich konnte mich sehr glücklich schätzen, eine Betreuung genießen zu dürfen, die wesentlich am Fortschritt meiner Arbeit interessiert war.

Abschließend möchte ich mich bei meiner Mutter und meiner Partnerin bedanken, die mir mein Studium durch ihre Unterstützung ermöglicht haben und stets ein offenes Ohr für mich hatten. Ohne euch wäre ich heute nicht an dieser Stelle!

Inhaltsverzeichnis

1	Problemstellung und Zielsetzung	9
1.1	Kontext, Motivation, Stand der Technik.....	9
1.2	Forschungsfragen.....	10
1.3	Zielsetzung	10
1.4	Nicht-Ziele.....	10
1.5	Lösungsansatz.....	10
1.6	Projektressourcen.....	11
1.6.1	Am Projekt beteiligten Personen & Unternehmen/Partner.....	11
1.6.2	Projektrelevante Ressourcen und deren Sicherstellung	11
1.7	Aufbau und Struktur der Arbeit.....	11
2	Theoretische Grundlagen.....	12
2.1	Relevanz von Python im Maschinenbau	12
2.2	Einfluss von KI-Tools auf Softwarequalität	16
2.3	Definition von Softwarequalität.....	18
2.3.1	Prozess- vs. Produktqualität.....	20
2.3.2	Innere vs. äußere Qualität.....	22
2.4	Qualitätsmodelle für Softwaresysteme	25
2.4.1	Boehm	26
2.4.2	McCall.....	27
2.4.3	FURPS (+)	29
2.4.4	Goal Question Metric	31
2.4.5	Dromey	33
2.4.6	ISO/IEC 9126.....	35
2.4.7	ISO/IEC 25010.....	36
2.4.8	ISO/IEC 25059.....	39
2.4.9	Arc 42 Q42.....	40
2.4.10	Praxiseinsatz	42
2.4.11	Fazit.....	45
2.5	Metriken für Softwaresysteme	47
2.5.1	Raw Metrics	48
2.5.2	Cyclomatic Complexity.....	49
2.5.3	Halstead Complexity	52

2.5.4	Maintainability Index	53
2.6	Allgemeine Software-Prinzipien	56
2.6.1	The Reason It All Exists [62]	56
2.6.2	KISS (Keep It Short & Simple) [64]	57
2.6.3	Maintain the Vision [61].....	57
2.6.4	What You Produce, Others Will Consume [65].....	58
2.6.5	YAGNI (You Aren't Gonna Need It) [66].....	58
2.6.6	Plan Ahead for Reuse [61]	59
2.6.7	Think [61].....	59
2.7	Large Language Model	60
2.7.1	Definition.....	60
2.7.2	Funktionsweise	61
2.7.3	Auswahl eines geeigneten Modells	66
2.7.4	Fine-tuning.....	67
2.8	Prompt Engineering	68
2.8.1	Best Practices für effektive Prompts.....	70
2.8.2	Bedeutung des Eingabeformats	73
3	Methodik	75
3.1	Clean-Code-Leitfaden für die Python-Entwicklung	75
3.1.1	General Pythonic Practices.....	77
3.1.2	Refactoring	81
3.1.3	Main Function and Main Name Idiom.....	87
3.1.4	Naming Conventions.....	90
3.1.5	Type Hints.....	102
3.1.6	Importing Modules	106
3.1.7	Docstrings.....	111
3.1.8	Formatting & spacing	124
3.2	Automatisierte Anwendung durch LLM-Prompting	125
3.2.1	Fallstudie 1	128
3.2.2	Fallstudie 2	131
3.2.3	Fallstudie 3	134
4	Ergebnis und Diskussion.....	137
4.1	Ergebnis	137

4.1.1	Forschungsfrage 1	137
4.1.2	Forschungsfrage 2	138
4.2	Diskussion	158
4.3	Weitere Vorgehensweise	159
	Literaturverzeichnis	162
	Abbildungsverzeichnis.....	173
	Tabellenverzeichnis.....	176
	Dokumentationstabelle KI-basierte Hilfsmittel.....	178
	Abkürzungsverzeichnis.....	179
	Anhang A: guideline.md	180
	Anhang B: optipy.py	208
	Anhang C: metrics_analyzer.py	219
	Anhang D: optipy-workflow.yml	225
	Anhang E: case_study_1.py	227
	Anhang F: case_study_2.py	228
	Anhang G: case_study_3.py	230
	Anhang H: case_study_results.csv.....	236
	Anhang I: GitHub-Repository.....	238

1 Problemstellung und Zielsetzung

1.1 Kontext, Motivation, Stand der Technik

In der heutigen schnelllebigen Softwareentwicklung spielt die Qualität eines Programmcodes eine sehr wichtige Rolle. Dies gilt insbesondere dann, wenn mehrere Personen in einem Softwareprojekt involviert sind. Damit eine kollaborative Arbeit im Team sichergestellt werden kann, sind Eigenschaften wie Lesbarkeit, Wartbarkeit oder Anpassbarkeit wichtiger denn je.

In den letzten Jahren hat sich Python, durch seine einfache Syntax, der breiten Masse an von Nutzern zur Verfügung gestellten Bibliotheken und durch seine Vielseitigkeit als eine der beliebtesten Programmiersprachen etabliert. Doch diese Beliebtheit bringt auch Probleme mit sich. Da Python häufig von Einsteigern oder Querschnittsfeldern als erste Programmiersprache gewählt wird und somit der erste Kontakt mit der Softwareentwicklung ist, streckt sich die Codequalität von kritisch zu exzellent.

Der einfache Zugriff zu öffentlichen Large Language Modellen, wie ChatGPT, und Coding-Assistenten, wie GitHub Copilot, ermöglicht zwar einen einfacheren Zugang zu Informationen, insbesondere in der Softwareentwicklung, und eine Produktivitätssteigerung beim Erstellen, Analysieren und Optimieren von Software-Applikationen, doch mit einem großen Preis. Die Hinterfragung des erstellten Outputs in Hinsicht auf etablierte Coding-Standards und Clean-Code-Prinzipien bleibt meist zu kurz. Dies zieht nicht nur bei unerfahrenen, sondern auch bei fortgeschrittenen Entwicklern eine Inkonsistenz mit sich, welche einen negativen Einfluss auf die Codequalität hat.

Doch eine unzureichende Codequalität kann erhebliche finanzielle Auswirkungen, Produktivitätseinbußen, Sicherheitsrisiken oder gar technische Gebrechen verursachen. Um diese Probleme vorzubeugen und damit qualitativ hochwertige Software zu gewährleisten, ist die Nutzung von Clean-Code-Prinzipien entscheidend. Diese Masterarbeit beschäftigt sich deshalb mit der Entwicklung und Anwendung von Clean-Code-Prinzipien in der Python-Entwicklung. Das Ziel ist es aus diesen Erkenntnissen einen pragmatischen Leitfaden zu erstellen, welcher sowohl Einsteigern als auch fortgeschrittenen Entwicklern einen einfachen Zugang zur Sicherstellung von Software mit hoher innerer Qualität bieten soll. Darüber hinaus soll eine Python-Applikation entwickelt werden, welche durch die Nutzung von externen LLMs eine automatisierte Anwendung des Leitfadens auf ein gegebenes Python-File ermöglicht und dadurch dessen innere Softwarequalität verbessert.

Insgesamt ist es also Ziel dieser Arbeit, einen umfassenden Überblick über das Thema Softwarequalität zu vermitteln und ein Tool zu entwickeln, welches eine automatisierte Anwendung von Clean-Code-Prinzipien in der Python-Entwicklung ermöglicht. Dies soll Entwickler:innen helfen, fundierte und effiziente Maßnahmen zur Optimierung des Entwicklungsprozesses zu ergreifen, um letztlich hochwertige Software zu gewährleisten.

1.2 Forschungsfragen

Um nun aus der oben angeführten Problemstellung einen nutzbaren Mehrwert ziehen zu können, ist die Beantwortung folgender Forschungsfragen essenziell:

1. Welche konkreten Maßnahmen können in Python getroffen werden, um die innere Softwarequalität zu verbessern, und wie lassen sich diese Maßnahmen in einem praxisnahen Clean-Code Leitfaden strukturiert zusammenfassen?
2. Wie kann die automatisierte Anwendung des entwickelten Clean-Code Leitfadens durch ein Large Language Model technisch umgesetzt werden, und inwiefern ist ein solcher Ansatz praktikabel bzw. wirtschaftlich sinnvoll?

1.3 Zielsetzung

Auf Basis der Fragestellungen (Kapitel 1.2.) ergibt sich folgende Zielsetzung für diese Arbeit:

1. Erarbeitung und Erläuterung von Clean-Code-Prinzipien zur systematischen Verbesserung der inneren Softwarequalität.
2. Entwicklung eines Python-Programmes zur automatisierten Anwendung des entwickelten Clean-Code-Leitfadens unter Verwendung eines Large Language Models.

1.4 Nicht-Ziele

Nicht-technische Faktoren, wie Teamdynamik oder Unternehmenskultur, sollen in dieser Arbeit nicht berücksichtigt werden.

1.5 Lösungsansatz

Um nun aus den oben angeführten Forschungsfragen die grundlegenden Ideen erläutern zu können, ist die Einteilung der Arbeit in folgende Lösungsansätze unumgänglich:

Erarbeitung und Erläuterung von Clean-Code-Prinzipien zur systematischen Verbesserung der inneren Softwarequalität

1. Erläuterung der relevantesten Clean-Code-Prinzipien in Python
2. Strukturierte Zusammenfassung der Clean-Code-Prinzipien in einer Markdown-Datei

Entwicklung eines Python-Programmes für die automatisierte Anwendung des entwickelten Clean-Code-Leitfadens unter Einbeziehung eines Large Language Models, ergänzt durch eine Bewertung des Modells anhand von unterschiedlichen Fallstudien

1. Entwicklung eines Python-Programmes zur automatisierten Anwendung des entwickelten Clean-Code-Leitfadens unter Verwendung von LLMs der Anbieter Anthropic, Google, OpenAI und xAI
2. Bewertung der verwendeten Modelle anhand von unterschiedlichen Fallstudien in Bezug auf Anzahl an Regelverstößen, Optimierungskosten und Optimierungsdauer.

1.6 Projektressourcen

1.6.1 Am Projekt beteiligten Personen & Unternehmen/Partner

Beteiligte	Typ
Philipp Gasser, BSc	Autor
FH-Prof. PD DI Dr. techn. Maximilian Lackner, MBA	1. Begutachter
Mag. Dr. Markus Pak	2. Begutachter

1.6.2 Projektrelevante Ressourcen und deren Sicherstellung

Ressource	Sicherstellung
Python	Gratis-Download
Anthropic API-Key	Eigenkapital
Google API-Key	Eigenkapital
OpenAI API-Key	Eigenkapital
Xai API-Key	Eigenkapital

1.7 Aufbau und Struktur der Arbeit

Folgende Abbildung simplifiziert den Aufbau dieser Arbeit und hebt die wesentlichen Schritte von den theoretischen Grundlagen über die methodische Umsetzung bis hin zur Validierung der Ergebnisse heraus:

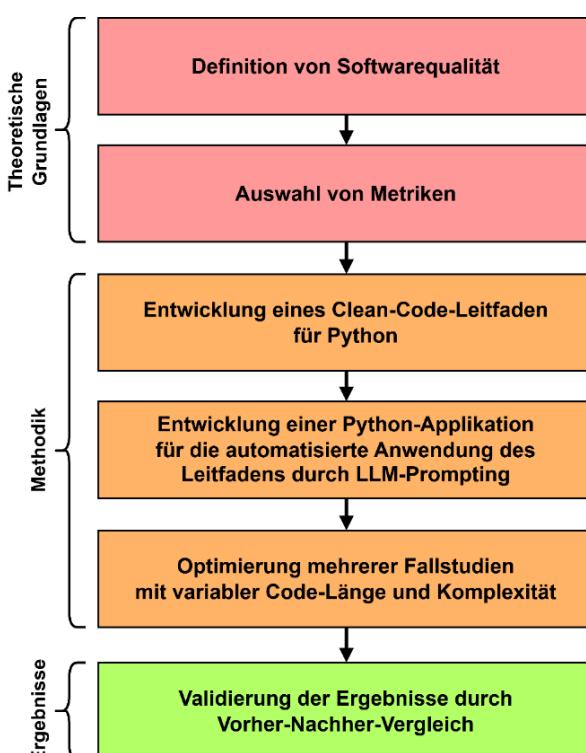


Abbildung 1: Aufbau dieser Arbeit

2 Theoretische Grundlagen

Bevor ein Tool zur Optimierung des Quellcodes in Python entwickelt werden kann, ist es wichtig, die theoretischen Grundlagen im Rahmen einer Literaturrecherche zu erläutern. Aufgrund der breiten Masse an zur Verfügung stehenden Literatur, wird die Literaturrecherche ein fundamentaler Teil dieser Arbeit sein. Nach der Diskussion des Begriffes „Softwarequalität“, einer Auflistung sämtlicher Qualitätsmodelle bzw. Metriken und relevanter Clean-Code-Prinzipien, soll schließlich ein automatisiertes Optimierungstool entwickelt werden.

2.1 Relevanz von Python im Maschinenbau

Der PYPL (PopularitY of Programming Language) Index basiert auf der Häufigkeit der Google-Suchen nach Tutorials der jeweiligen Programmiersprache. Der Index wurde im Jahre 2004 von P. Carbonelle [1] erstellt und nutzt die zugrundeliegenden Rohdaten von Google Trends. Die Hypothese besagt, dass die Popularität einer Programmiersprache mit der Häufigkeit an Suchanfragen auf Google nach sprachspezifischen Tutorials korreliert.

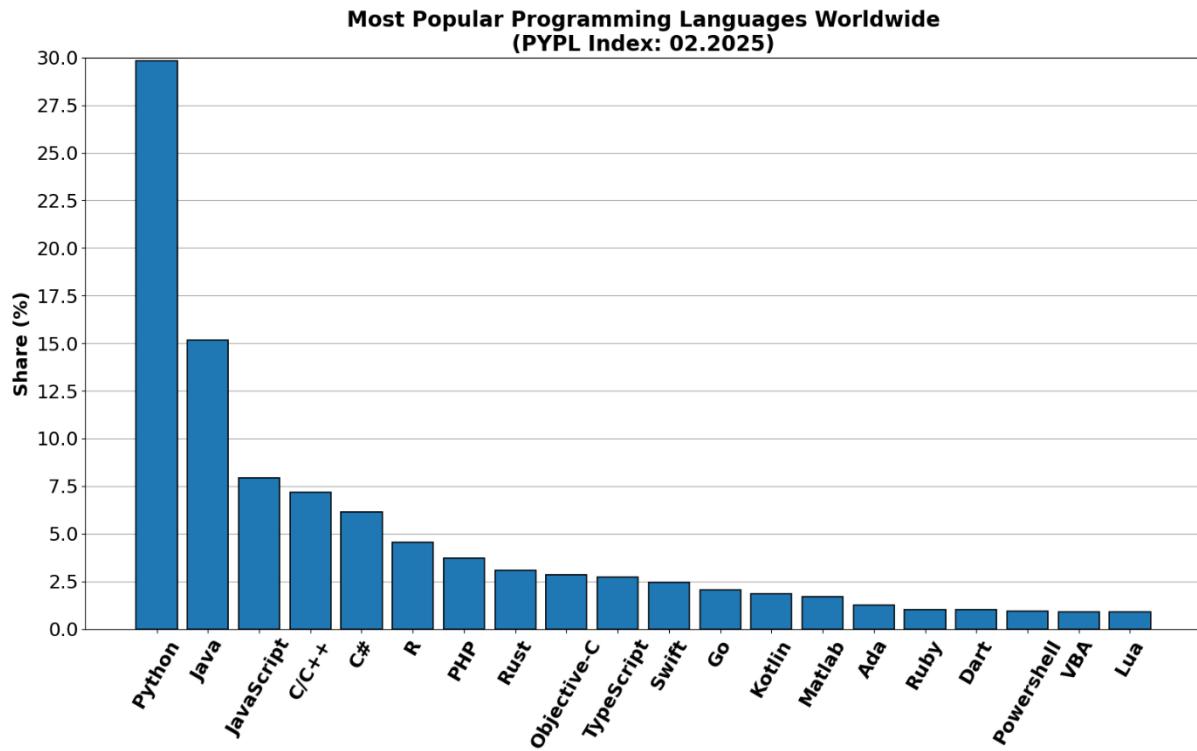


Abbildung 2: Beliebtheit von Programmiersprachen basierend auf dem PYPL-Index (Februar 2025) [1]
(eigene Abbildung)

Da der PYPL-Index das Schlagwort „Tutorial“ beinhaltet könnte argumentiert werden, dass diese Popularität auf ein Lerninteresse von Neueinsteigern hindeutet.

Damit aber auch die Beliebtheit von Python hinsichtlich des Industrieeneinsatzes bewertet werden kann, ist die Erläuterung des TIOBE (The Importance of Being Earnest) Index notwendig. Der Index wurde im Jahre 2001 von dem niederländischen IT-Unternehmen TIOBE Software BV [2] entwickelt und nutzt eine Vielzahl an Datenquellen, darunter beispielsweise Google, Wikipedia, Amazon und Bing. Eine detaillierte Erklärung der Berechnung ist der Dokumentation [3] zu entnehmen.

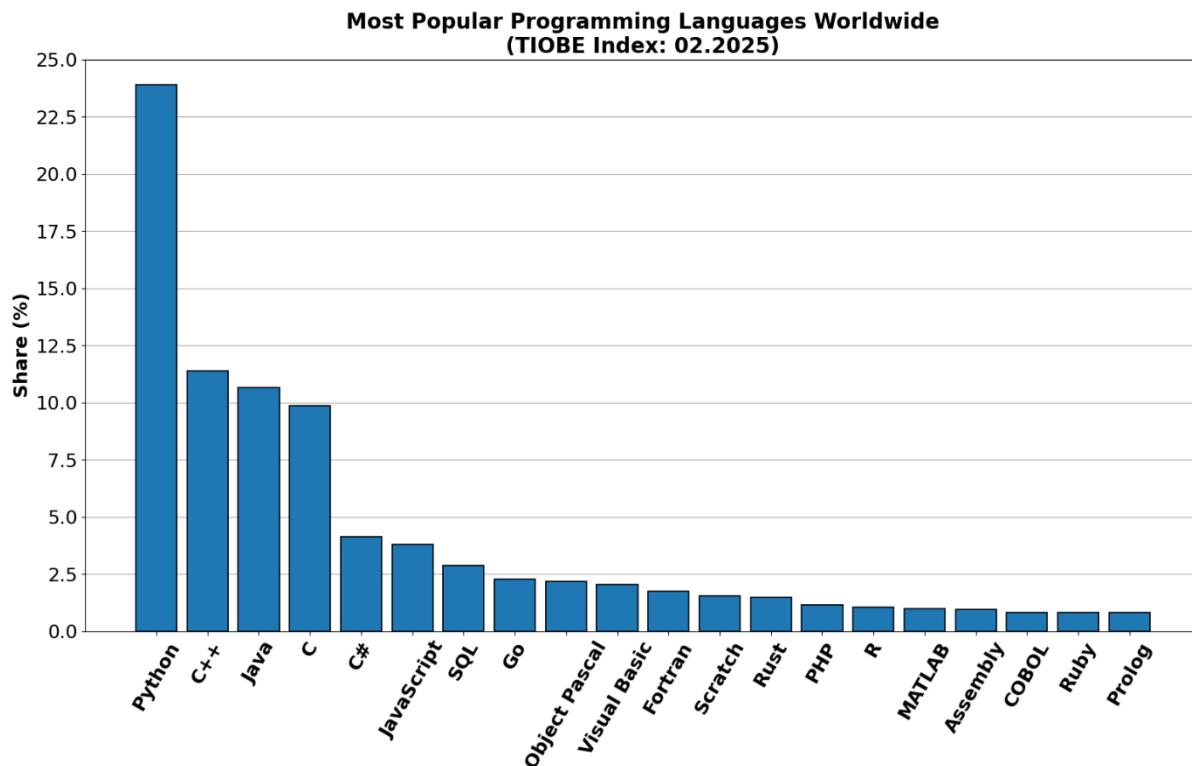


Abbildung 3: Beliebtheit von Programmiersprachen basierend auf dem TIOBE-Index (Februar 2025) [2] (eigene Abbildung)

Vergleich der Popularität von Python mit Matlab:

Um den Bezug zum Maschinenbau zu verstärken, ist ein Vergleich zwischen Matlab und Python essenziell. Einige Zeit war Matlab die bevorzugte Programmiersprache für ingenieurwissenschaftliche Berechnungen und Simulationen. Dies ist auf die damalige Bereitstellung umfangreicher Toolboxen (z.B.: Simulink), vordefinierter Funktionen und optimierter Algorithmen, zurückzuführen. Dadurch war es möglich komplexe Probleme verhältnismäßig effizient zu lösen, ohne die zugrundeliegenden Programme selbst erstellen zu müssen.

Doch mit der wachsenden Community und der damit verbundenen Erweiterung bzw. Verbesserung der öffentlich zugänglichen Bibliotheken gewann Python an Relevanz. Darüber hinaus ist Python mit seiner Open-Source-Natur eine wirtschaftlich attraktive Alternative zu dem kostenpflichtigen Lizenzsystem von Matlab. Die einfache Syntax, die universelle Anwendbarkeit und die Popularität machen Python zu einer vielseitigen Allzwecksprache.

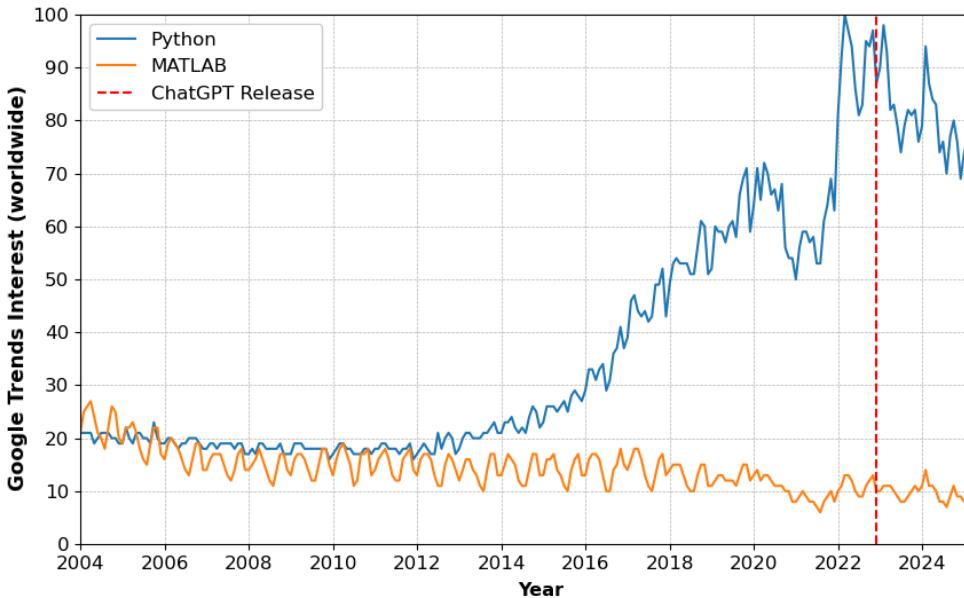


Abbildung 4: Google-Suchinteresse an Python und MATLAB weltweit (2004–2025) [4]
(eigene Abbildung)

Damit eine objektive Bewertung gewährleistet wird, wird neben der Google-Suche auch die Relevanz auf Stack Overflow, einer der bekanntesten Entwickler-Communitys, herangezogen:

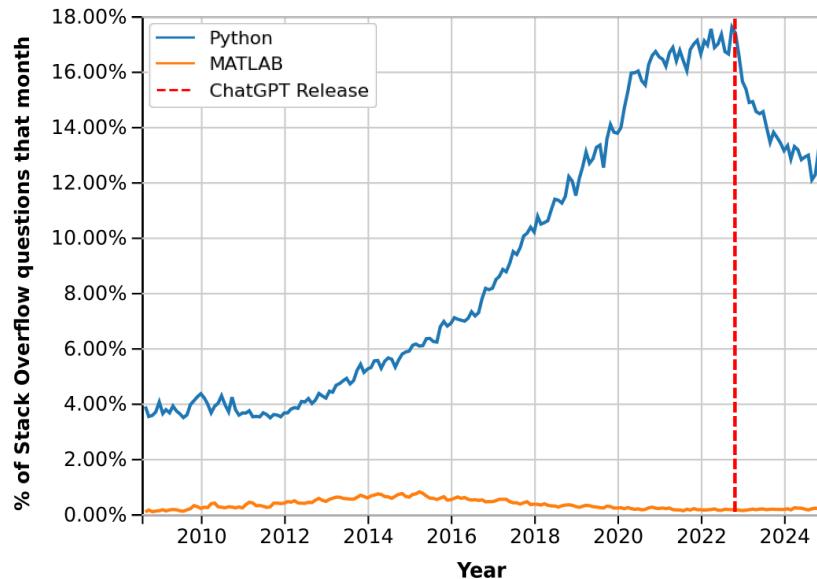


Abbildung 5: Anteil der Stack Overflow-Fragen zu Python und MATLAB pro Monat (2008–2025) [5]
(eigene Abbildung)[1]

Der plötzliche Einbruch an Fragen zu Python auf Stack Overflow ist sehr wahrscheinlich auf die Veröffentlichung des ersten kostenfreien, KI-gestützten Chatbots ChatGPT am 30. November 2022 [6] zurückzuführen.

Zusammengefasst kann untermauert werden, dass sich Python höchstwahrscheinlich durch seine einfache Syntax, seine vielseitige Anwendbarkeit und seine kostenfreie Nutzung als eine der führenden Programmiersprachen etabliert hat.

Einsatzgebiete von Python im Maschinenbau:

Um die Relevanz von Python im Maschinenbau zu verdeutlichen, werden einige Anwendungsbereiche erläutert:

Tabelle 1: Einsatzgebiete von Python im Maschinenbau

Einsatzgebiet	Software / Technologie	Python-Bibliothek
Finite-Elemente-Methode (FEM)	Ansys Mechanical Abaqus CAE Ansys LS-DYNA	PyMechanical Abqpy PyDYNA
Strömungsmechanik (CFD)	Ansys Fluent OpenFOAM	PyFluent PyFoam
Mehrkörpersimulation (MKS)	RecurDyn	Interne API
Geometrieverarbeitung (CAD)	CadQuery FreeCAD	CadQuery FreeCAD
Datenverarbeitung	/ / /	NumPy Pandas SciPy
Datenvisualisierung	/ / / /	Matplotlib Seaborn Plotly ydata-profiling
Machine Learning	/ / / /	Scikit-learn TensorFlow PyTorch PyCaret
Bildverarbeitung (z.B.: Qualitätskontrolle)	/ /	OpenCV scikit-image
Roboterprogrammierung	RoboDK ABB RobotStudio	RoboDK Interne API
Internet of Things (IoT) (z.B.: Predictive Maintenance)	MQTT OPC-UA Modbus	Paho-MQTT Python OPC-UA PyModbus
Produktionsplanung	Visual Components	Interne API

Durch die steigende Bedeutung von digitalen Lösungen verschwimmen die Grenzen zwischen dem klassischen Maschinenbau und anderen Disziplinen, wie der Softwareentwicklung, zunehmend.

2.2 Einfluss von KI-Tools auf Softwarequalität

Das Unternehmen Alloy.dev untersucht im Rahmen seines Projektes „Gitclear“ den Einfluss von KI-gestützten Coding-Assistenten auf Softwarequalität, um Prognosen zur Wartbarkeit zukünftiger Softwaresystemen zu erstellen. Dazu sammeln sie, wie der Name bereits vermuten lässt, Daten von Git-Repositories von privaten Tech-Unternehmen (z.B.: NextGen Health, Bank of Georgia) und Open-Source-Projekten (z.B.: Google Chromium, Facebook React, Microsoft Visual Studio Code).

In der diesjährigen Ausgabe (2025) [7] analysierten sie auf Basis von 211.000.000 Codezeilen sämtliche Codeänderungen, die im Zeitraum von Januar 2020 bis Dezember 2024 vorgenommen wurden. Die Auflistung sämtlich verwendeter Repositories ist [7] auf Seite 24 zu entnehmen. Dabei wurden die Codeänderungen folgendermaßen kategorisiert:

- **Added Code:** Codezeilen, die neu hinzugefügt wurden.
- **Deleted Code:** Codezeilen, die entfernt und mindestens zwei Wochen nicht wieder hinzugefügt wurden.
- **Updated Code:** Codezeilen, die bereits existierten und minimal geändert wurden (max. drei Wörter).
- **Moved Code:** Codezeilen, die ohne inhaltliche Änderung in eine andere Datei oder Funktion verschoben wurden.
- **Copy/pasted Code:** Codezeilen, die mehrfach in Dateien oder Funktionen eingefügt wurden.
- **Find/replaced Code:** Codezeilen, die an mindestens drei Stellen einheitlich durch einen neuen String ersetzt wurden.
- No-Operation-Änderungen im Sinne der Formatierung werden nicht berücksichtigt.

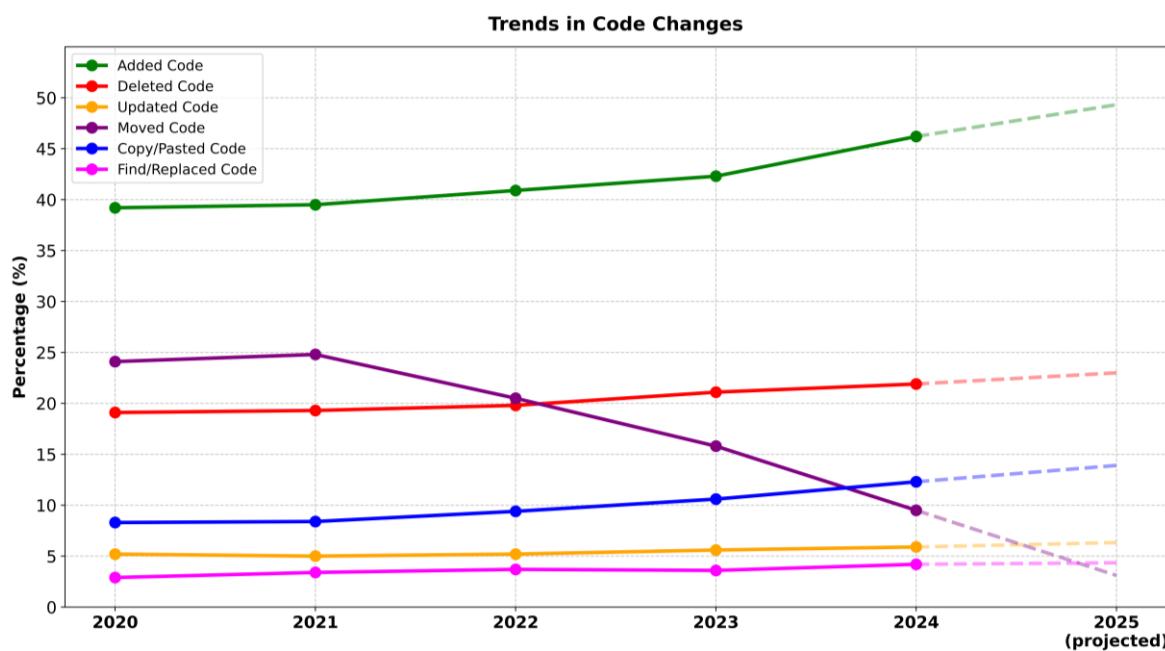


Abbildung 6: Relative Verteilung der Code-Änderungen pro Jahr, einschließlich einer Prognose für 2025 von GitClear [7] (eigene Abbildung)

Beschreibung der Ergebnisse

Die Abbildung zeigt, dass es zwischen den Jahren 2020 und 2021 keine signifikanten Veränderungen gab. Zwischen den Jahren 2021 und 2024 sind jedoch deutliche Trends erkennbar (sortiert nach relativer Änderung):

- Der Anteil an „Moved Code“ ist von 24,1% auf 9,5% gefallen.
(Relative Änderung von -60,6%)
- Der Anteil an „Copy/Pasted Code“ verzeichnete einen Anstieg von 8,4% auf 12,3%.
(Relative Änderung von +46,4%)
- Der Anteil an „Added Code“ ist von 39,5% auf 46,2% gestiegen.
(Relative Änderung von +17,0%)
- Der Anteil an „Deleted Code“ ist von 19,3% auf 21,9% gesunken.
(Relative Änderung von +13,5%)

Interpretation der Ergebnisse

Der signifikante Rückgang des Anteils an „Moved Code“ in Kombination mit dem markanten Anstieg des Anteils an „Copy/Pasted Code“ deutet darauf hin, dass bestehende Strukturen seltener faktorisiert und stattdessen vermehrt dupliziert werden. Darüber hinaus könnte der vergleichsweise moderate Zuwachs an „Added Code“ und „Deleted Code“ bedeuten, dass bestehender Code seltener überarbeitet wird und stattdessen durch neuen Code ersetzt wird.

Durch die zunehmende Duplizierung und abnehmende Refaktorisierung steigt das Risiko an Inkonsistenzen und erhöhtem Wartungsaufwand. Diese Umstände können erhebliche Auswirkungen auf die Codequalität und in weiterer Folge auf die Wartbarkeit haben.

Doch wodurch ergibt sich dieser Trend überhaupt? Der erste weitgehend anerkannte öffentliche KI-gestützte Coding-Assistent ist GitHub Copilot wurde am 29. Juni 2021 [8] in einer Zusammenarbeit zwischen GitHub und OpenAI veröffentlicht. Dieser Zeitpunkt steht in Konfluenz mit den beobachteten Änderungen seit dem Jahr 2021. Dass KI-gestützte Coding-Assistenten tatsächlich eingesetzt werden, zeigt der Google DORA Report 2024 [9]. Der Bericht hebt hervor, dass das Schreiben von neuem Code mit 74,9% die am häufigsten durchgeführte Aufgabe der Befragten ist. Dies untermauert die Relevanz solcher Tools im heutigen Softwareentwicklungsprozess.

Basierend auf der für das Jahr 2025 erstellten Prognose, die mit einer linearen Regression erstellt wurde, deutet es darauf hin, dass sich die bisherigen Entwicklungen weiter intensivieren werden.

2.3 Definition von Softwarequalität

Der Begriff „Softwarequalität“ wird in der Literatur auf verschiedene Arten definiert:

- IEEE 1074-2006: *“The degree to which a system, component, or process meets specified requirements and/or the customer/user needs and expectations.”* [10]
- ISO/IEC 9126-1:2001: *“The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs.”* [11] (S. 20)
- Pressman: *“An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.”* [12] (S. 414)

Diese unterschiedlichen Definitionen bzw. Perspektiven verdeutlichen, dass es keine universellen Kriterien zur Interpretation von Softwarequalität gibt. Sie hängt nämlich einerseits von den spezifischen Anforderungen der Kunden und andererseits von den Erwartungen der involvierten Stakeholder ab.

Doch warum entsteht in erster Linie überhaupt eine Diskussion über Softwarequalität? Der vermutlich weitverbreitetste Grund, warum Entwickler mangelhaften Quellcode schreiben, ist die vermeintlich schnelle und einfache Implementierung. Diese möge in manchen Individualfällen kurzfristig auch kostengünstiger sein, doch selbst bei alleiniger Arbeit an einem privaten Projekt (ohne Team) können die meisten Entwickler ihren eigenen Quellcode nach einiger Zeit selbst kaum noch nachvollziehen. Dies deutet zudem darauf hin, dass hochwertiger Quellcode nicht nur im Rahmen eines Teams, sondern auch für die Weiterentwicklung eines Entwicklers selbst essenziell ist.

Dass Softwarequalität auch eine wichtige wirtschaftliche Rolle spielt, bestätigte das IT-Unternehmen Digital, Inc., welches die relativen Kosten für die Korrektur von Fehlern in den verschiedenen Softwareentwicklungsphasen im Rahmen einer Fallstudie [13] auf Basis eines Papers von Boehm und Basili [14] visualisierte. Da die Kosten für die Korrektur eines Fehlers von vielen Faktoren abhängen und stark variieren, verwendeten Boehm und Basili einen relativen Kostenfaktor, um die Unterschiede in den jeweiligen Softwareentwicklungsphasen zu verdeutlichen. Diese Kostenfaktoren wurden von Digital, Inc. anhand eines Fallbeispiels mit zur Zeit der Erstellung der Grafik praxisüblichen Kosten dargestellt:

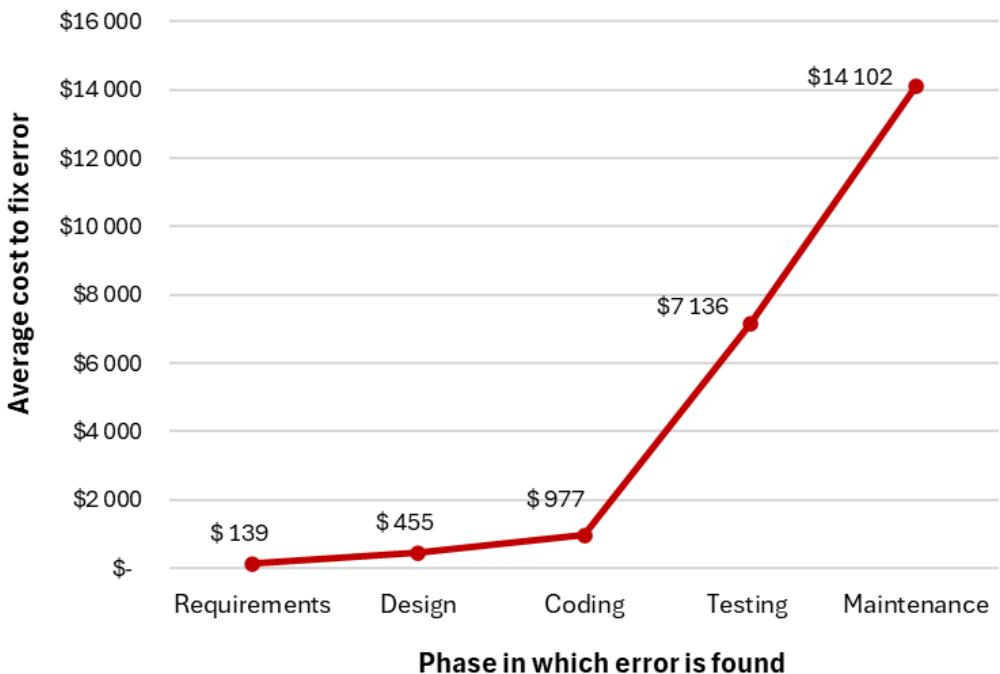


Abbildung 7: (Nicht inflationsbereinigte) Relative Kosten für die Korrektur eines Fehlers [13][14] (eigene Darstellung)

Es zeigt sich, dass die Kosten für Fehlerkorrekturen im Laufe des Softwareentwicklungsprozesses exponentiell ansteigen. Desto früher ein Fehler identifiziert und behoben werden kann, desto geringer die Unkosten und desto größer die Kosteneinsparungen. Diese Feststellung verdeutlicht, dass die Qualität, insbesondere die innere Qualität einer Software, eine Schlüsselrolle spielt. Präventive Maßnahmen, wie die gründliche Erarbeitung von sprachenüblichen Best-Practices, verbessern nicht nur die Codequalität, sondern können auch signifikante Kosteneinsparungen erlauben.

Jedoch gibt es Sonderfälle, in denen hochwertiger Quellcode nicht zwingend erforderlich ist bzw. sogar als ineffizient betrachtet werden könnte. Diese lauten wie folgt [15]:

- *Einfache Skripte* (für den einmaligen Gebrauch)
- *Proof of Concepts*
- *Legacy-Projekte* (veraltete Projekte, die kurzfristig gewartet und bald abgelöst werden)
- *Hackathons & Programmierwettbewerbe*

Alle diese Fälle haben gemeinsam, dass es sehr unwahrscheinlich ist, dass der Code nochmal verwendet wird und auch grundsätzlich nicht für die langfristige Wartung geplant ist. Andererseits kann es natürlich passieren, dass aus „einfachen“ Skripten unerwartet viel größere Projekte entstehen, in welchen die Wartbarkeit doch eine wichtige Rolle spielt. Darüber hinaus kann es der Fall sein, dass innerhalb eines Unternehmens ein Projekt plötzlich an ein anderes Team übergeben werden muss. Hier ist eine saubere Codebasis Grundvoraussetzung, um eine reibungslose Übergabe und spätere Wartung zu ermöglichen.

Zusammengesagt kann daher gesagt werden, dass es empfehlenswert ist, grundlegende Best Practices anzuwenden, selbst in Projekten, die vorerst keine langfristigen Pläne verfolgen.

2.3.1 Prozess- vs. Produktqualität

Damit Softwarequalität besser nachvollzogen werden kann, ist es wichtig zwischen Prozess- und Produktqualität zu unterscheiden. Im Rahmen dieser Arbeit wird, sofern nicht anders angegeben, der Begriff „Softwarequalität“ im Sinne der Produktqualität verwendet. Dies wird in den nachstehenden Kapiteln anhand der ISO-Standards besser ersichtlich.

Prozessqualität

Die Prozessqualität definiert die Qualität des Herstellungsprozesses eines Produktes. In Projekten ist es wichtig, Prozesse gezielt zu spezifizieren und kontinuierlich zu optimieren. [16] Dennoch gilt es zu beachten, dass eine Prozessorientierung nicht zwangsläufig zu einer Qualitätsbewertung führt, die unabhängig vom Produkt selbst ist. [17]

Produktqualität

Die Produktqualität beschreibt die Qualität eines fertig entwickelten Produkts. Generell kann gesagt werden, dass die Produktqualität ein Maß dafür ist, wie gut die Anforderungen und Erwartungen der Stakeholder, insbesondere der Kunden, erfüllt werden. Das grundlegende Ziel eines Softwareprojektes ist es, ein qualitativ hochwertiges Produkt zu erstellen. [16]

Laut Darvin [18] gibt es fünf Ansätze, um Produktqualität zu betrachten:

1. Transzendenten Sicht

Die transzendenten Sicht basiert auf der philosophischen Diskussion von Plato über Schönheit und beschreibt Qualität als „angeborene Exzellenz“, welche nicht konkret definiert, sondern nur durch *Erfahrung* erkannt werden kann. Diese Sicht stellt Qualität als ein Ideal dar, welchem ein Produkt entsprechen sollte.

2. Produktorientierte Sicht

Die produktorientierte Sicht erläutert Qualität als exakt definierbare und messbare Größe. Die Beurteilung der Qualität erfolgt anhand von festgelegten Produkteigenschaften und daher auch *objektiv*. Dieser Ansatz wird häufig als Grundlage für Qualitätsmodelle bzw. Metriken verwendet.

3. Wertorientierte Sicht

Die wertorientierte Sicht beschreibt Qualität in Bezug zu Kosten bzw. Nutzen und entspricht vereinfacht gesagt, dem *Preis-Leistungs-Verhältnis* eines Produktes. Sie bietet eine nützliche Ergänzung zu den anderen vier Ansätzen.

4. Benutzerorientierte Sicht

Die benutzerorientierte Sicht betrachtet Qualität aus der *externen* Benutzerperspektive. Da jeder Kunde bzw. jeder Benutzer andere Bedürfnisse hat, erfolgt die Bewertung subjektiv und liegt daher „im Auge des Betrachters“.

Insbesondere Eigenschaften, wie Benutzbarkeit oder Zuverlässigkeit, spielen hier eine große Rolle.

5. Herstellerorientierte Sicht

Im Gegensatz zur benutzerorientierten Sicht, die eine externe Bewertung vorsieht, strebt die herstellerorientierte Sicht eine *interne* Bewertung an. Qualität entspricht hier der Einhaltung von spezifischen Anforderungen, und zielt darauf ab, Nachbesserungen von Fehlern zu vermeiden und damit verbundene Unkosten zu minimieren.

Zusammenhang zwischen Prozess- und Produktqualität

Während der Zusammenhang zwischen Prozess- und Produktqualität in klassischen Disziplinen, wie der Fertigungstechnik, meist eindeutig ist, ist dieser bei Entwicklungsprozessen, insbesondere in der Softwareentwicklung weniger klar. [17][19]

Dies wurde durch Capers Jones [20] im Jahre 2000 bestätigt, als er die Beziehung zwischen der Reifegradstufe eines Unternehmens im Capability Maturity Model (CMM) und der Fehleranzahl pro Funktionsblock analysierte.

Tabelle 2: Gelieferte Fehler pro Funktionsblock in Bezug zu CMM-Stufe [20]

CMM-Level	Minimum	Average	Maximum
1	0,150	0,750	4,500
2	0,120	0,624	3,600
3	0,075	0,473	2,250
4	0,023	0,228	1,200
5	0,002	0,105	0,500

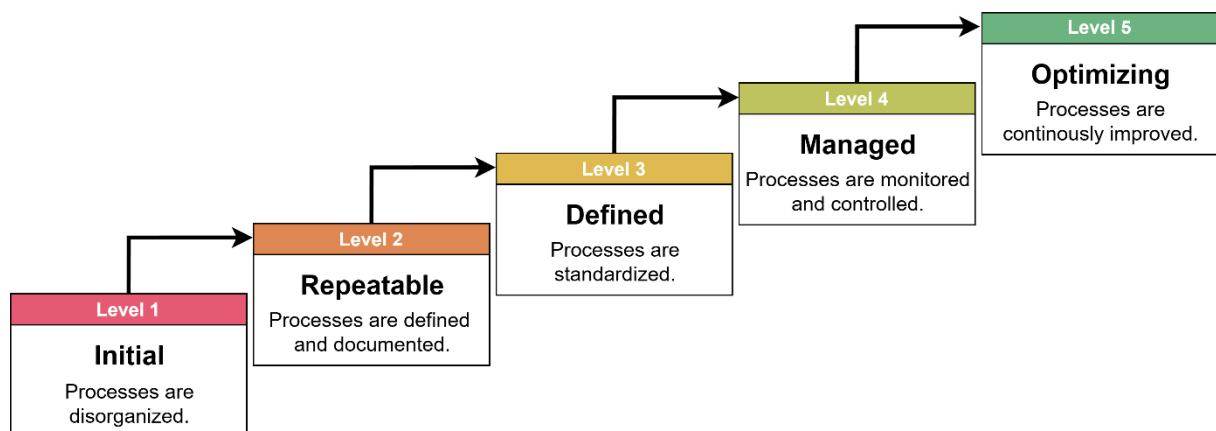


Abbildung 8: Aufbau eines Reifegradmodells (Capability Maturity Model)

Seine Ergebnisse zeigen, dass höhere Reifegradstufen (Level 5) zwar meist eine geringere Fehlerquote aufweisen, es aber auch Unternehmen auf niedrigen Reifegradstufen (Level 1) gibt, die eine bessere Produktqualität erreichen (siehe Tabelle 2). Er betont daher, dass neben der Prozessqualität stets eine unabhängige Überwachung des entstehenden Produktes zu erfolgen hat, um letztlich Produktqualität zu gewährleisten.

Zusammengefasst kann gesagt werden, dass in der Softwareentwicklung insbesondere Fokus auf die Produktqualität gelegt werden sollte.

2.3.2 Innere vs. äußere Qualität

Innere Qualität

Die innere Qualität bezieht sich auf die Eigenschaften, die sich mit dem Quellcode und der internen Funktionsweise einer Software beschäftigen. Insbesondere als Entwickler und im Rahmen dieser Arbeit ist die innere Qualität deshalb von großer Relevanz, weil sie maßgeblich beeinflusst, wie einfach und effizient eine Software gewartet, weiterentwickelt oder an neue Anforderungen angepasst werden kann. Eine hohe innere Qualität erleichtert solche Anpassungen und gestaltet den Entwicklungsprozess insgesamt effizienter und ressourcenschonender. [21]

Da der/die Kunde/-in die konkrete Implementierung nur in Ausnahmefällen sieht, möchte er/sie darin weder Zeit noch Geld investieren. Interessanterweise ist es jedoch oft genau die innere Qualität, die den Projekterfolg gewährleistet oder eben nicht. [22]

Ähnliches hat auch Fielding im Rahmen seiner Dissertation festgestellt: Ein Beispiel für den Erfolg von Produkten mit hoher innerer Qualität ist das Hypertext Transfer Protocol (HTTP), welches nicht erfolgreich war, weil es mehr konnte, sondern wegen der Art und Weise, wie es implementiert wurde. [23]

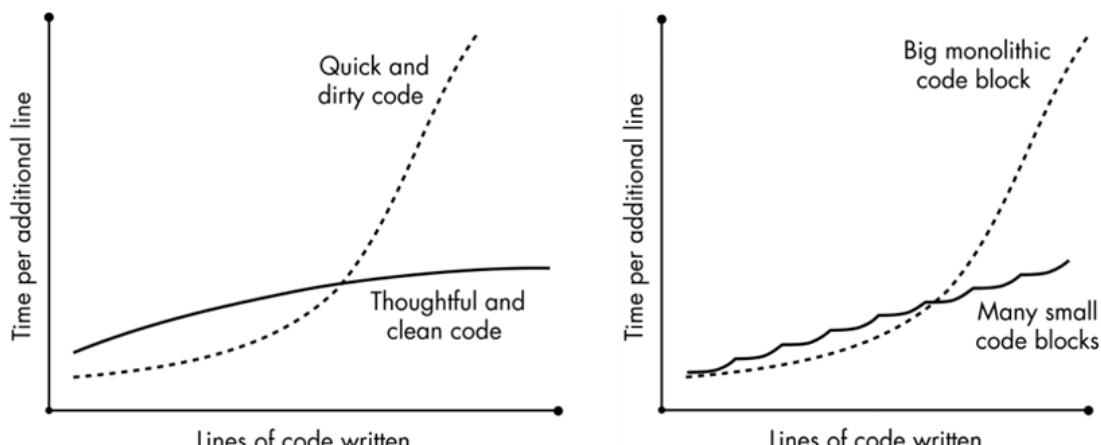


Abbildung 9: Einfluss der Codequalität (innere Qualität) auf den Entwicklungsprozess [24]

Wenn innere Softwarequalität vernachlässigt wird, beispielsweise bei der Implementierung von langen unverständlichen Codeblöcken, sammeln sich im Laufe der Entwicklung technische Verbindlichkeiten an. Diese wachsen mit jeder Erweiterung und führen unmittelbar dazu, dass künftige Anpassungen des Quellcodes fortlaufend aufwendiger werden. Der zusätzliche Aufwand sorgt auch für Unkosten, die die Wirtschaftlichkeit des Projektes gefährden können. Ab einer gewissen Anzahl an geschriebenen Codezeilen bzw. einem bestimmten Komplexitätsgrad, ist eine Weiterentwicklung der Software so umständlich, in einigen Fällen sogar unmöglich, dass eine komplette Neuentwicklung erforderlich ist. Um derartige Probleme

zu vermeiden und langlebige, wartungsfreundliche Softwarelösungen sicherzustellen, sollte die innere Softwarequalität bereits mit Beginn des Entwicklungsprozesses auf einem hohen Niveau sein. Dadurch können innere Qualitätsprobleme bereits frühzeitig erkannt und mit dementsprechenden Korrekturmaßnahmen versehen werden.

Eigenschaften für hohe innere Qualität sind unter anderem [16]:

- Gute Wartbarkeit
- Einfache Verständlichkeit
- Gute Lesbarkeit des Quellcodes
- Einfache Wiederverwendbarkeit
- Einfache Erweiterbarkeit

Äußere Qualität

Die äußere Qualität beschäftigt sich mit der Perspektive des Kunden, also derjenigen Person, die mit der fertig entwickelten Software interagieren bzw. arbeiten muss. Sie kann erst direkt gemessen werden, wenn das Produkt prototypenreif oder fertig entwickelt ist. [21]

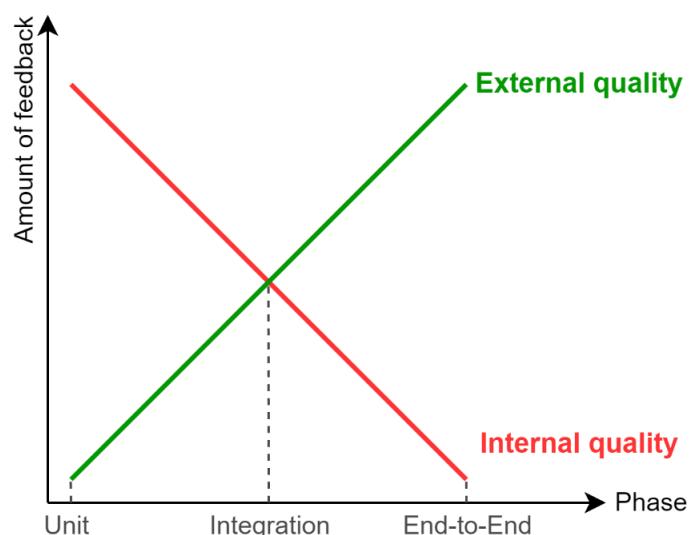


Abbildung 10: Relevanz der inneren und äußeren Qualität während den Testphasen [25]
(eigene Darstellung)

Es wird ersichtlich, dass die äußere Qualität nicht erst beim End-to-End-Testing, beispielweise in der Beta-Phase einer Software, sondern bereits in der Integrationsphase eine erstmalige Rolle spielt.

Eigenschaften für hohe äußere Qualität sind unter anderem [16][21]:

- Hohe Zuverlässigkeit
- Intuitive Bedienbarkeit
- Einfache Benutzbarkeit
- Hohe Robustheit
- Hohe Fehlertoleranz

Zusammenhang zwischen innerer und äußerer Qualität

Dimitris Stavrinoudis und Michalis Xenos haben im Rahmen ihrer Forschung [26] im Jahr 2008 einen Zusammenhang zwischen innerer und äußerer Softwarequalität feststellen können. Sie zeigten auf, dass Metriken zur Messung der inneren Softwarequalität eine zuverlässige Ersteschätzung für die äußere Softwarequalität bieten. Insbesondere während der Entwicklungsphase, also noch lange vor Lieferung der Software an den Kunden, bieten innere Qualitätsmetriken eine attraktive Form der Qualitätssicherung.

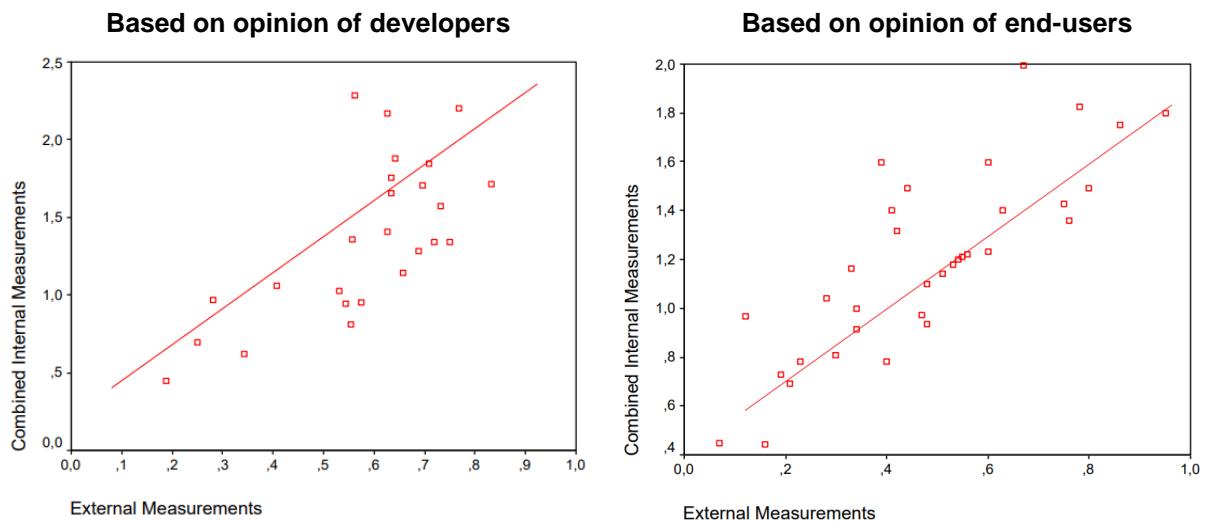


Abbildung 11: Zusammenhang zwischen innerer und äußerer Qualität nach [26]

Wichtig ist jedoch zu beachten, dass eine gute innere Qualität kein Garant für gute äußere Qualität ist. Ist beispielweise der Quellcode schön strukturiert, effizient und einfach wartbar, aber die Bedienoberfläche mangelhaft, wird der Benutzer unzufrieden sein. Gegenteiliges gilt ebenfalls: Eine niedrige innere Qualität muss keine niedrige äußere Qualität implizieren. Zum Beispiel kann es der Fall sein, dass es sich im Inneren um „Spaghetti Code“ handelt, welcher eine Lesbarkeit oder Wartbarkeit des Quellcodes erschwert oder gar unmöglich macht, der Benutzer aber mit der Software trotzdem zufrieden ist.

Zusammenfassend ist wichtig zu beachten, dass innere und äußere Qualität eng miteinander verbunden sind und sich gegenseitig ergänzen. Allgemein kann aber gesagt werden, dass eine gute innere Qualität, insbesondere bei großen, komplexen oder lang andauernden Projekten, in den meisten Fällen einen positiven Einfluss auf die äußere Qualität hat. Da sich diese Arbeit vorrangig mit der Entwicklung von Software beschäftigt, soll die Relevanz der inneren Qualität, also die Sicht des Entwicklers, im Fokus stehen.

2.4 Qualitätsmodelle für Softwaresysteme

Grundsätzlich gibt es keine standardisierten Kriterien für die Bewertung von qualitativ hochwertigem Quellcode. Dies zieht das Problem mit sich, dass jeder Entwickler, jedes Unternehmen oder jede Industrie ihre eigenen Qualitätseigenschaften individuell definiert. Damit dieses Problem gelöst wird und eine einheitliche standardisierte Definition von Kriterien gewährleistet werden kann, wurden sogenannte Qualitätsmodelle entwickelt. Qualitätsmodelle ermöglichen es, Qualitätsanforderungen systematisch zu modellieren, die Qualität von Softwaresystemen zu analysieren und zu überwachen, sowie die dafür geeigneten Metriken, Modelle und Bewertungsverfahren offenzulegen. Obwohl Qualitätsmodelle ständig angepasst und umstrukturiert werden, bieten sie einen relativ guten Überblick über die relevantesten Qualitätseigenschaften. Wie sich später zeigen wird, gibt es nicht "das" richtige Qualitätsmodell. Die Modelle sind nämlich zwar meist standardisiert, berücksichtigen die von Projekt zu Projekt maßgeblich variierenden individuellen Anforderungen jedoch nicht.

Im Rahmen dieser Arbeit werden die relevantesten Qualitätsmodelle erwähnt und näher erläutert:

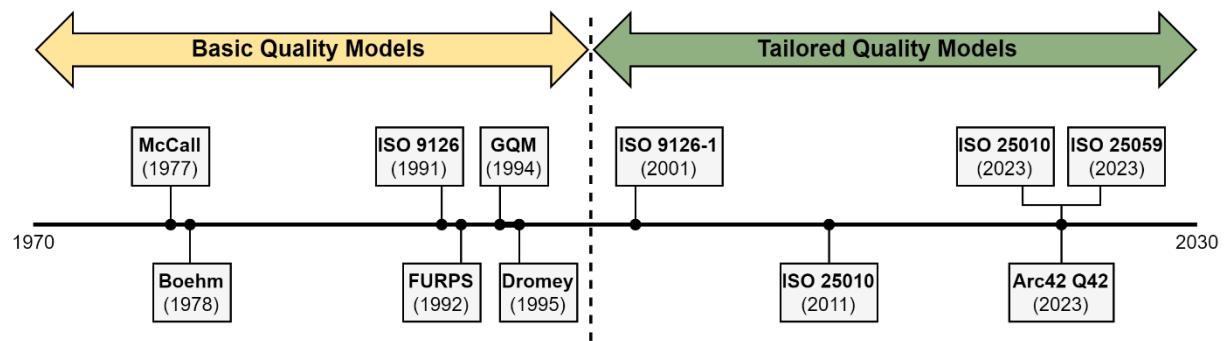


Abbildung 12: Zeitstrahl der Historie der relevantesten Qualitätsmodelle

Angemerkt sei hier jedoch, dass es neben den in diesem Kapitel erwähnten Qualitätsmodellen noch unzählige weitere nennenswerte Modelle gibt, beispielsweise:

- 1987 – Evans and Marciniak Quality Model
- 1988 – F. Deutsch & Wills Quality Model
- 2009 – Squale Model
- 2012 – Quamoco Model

Doch um den Rahmen dieser Arbeit nicht zu sprengen, werden sie nur am Rande erwähnt und nicht im Detail erläutert. Lediglich etablierte und weit verbreitete Qualitätsmodelle sollen ergänzend herangezogen werden, um ein grundlegendes Verständnis der Softwarequalität zu schaffen.

2.4.1 Boehm

Das Boehm Software Qualitätsmodell [27] beschreibt eine Hierarchie von Qualitätsmerkmalen, die die Qualität von Softwaresystemen systematisch analysiert. Im Gegensatz zu den heutigen ISO-Standards, weist das Boehm Modell eine flexible Struktur auf, weshalb den Hauptqualitätsmerkmalen mehrere Qualitätsteilmerkmale zugewiesen werden können. Das Modell ist nach Barry Boehm benannt und wurde im Jahre 1976 erstellt, also in einer Zeit, in welcher der Zugriff zu Computern hauptsächlich Unternehmen vorbehalten und nur in Ausnahmefällen auch Privatleuten gestattet war. Boehm war seiner Zeit weit voraus und erstellte das erste Modell, welches die Qualität eines Systems in drei Ebenen einteilte.

Der Grundgedanke des Modells ist die Beantwortung folgender drei Fragen bzw. Kategorien:

1. As-Is Utility

Wie gut (einfach, verlässlich und effizient) kann das vorhandene System genutzt werden?

2. Maintainability

Wie einfach kann das System verstanden, modifiziert, gewartet und getestet werden?

3. Portability

Kann das System auch genutzt werden, wenn auf eine andere Umgebung gewechselt wird?

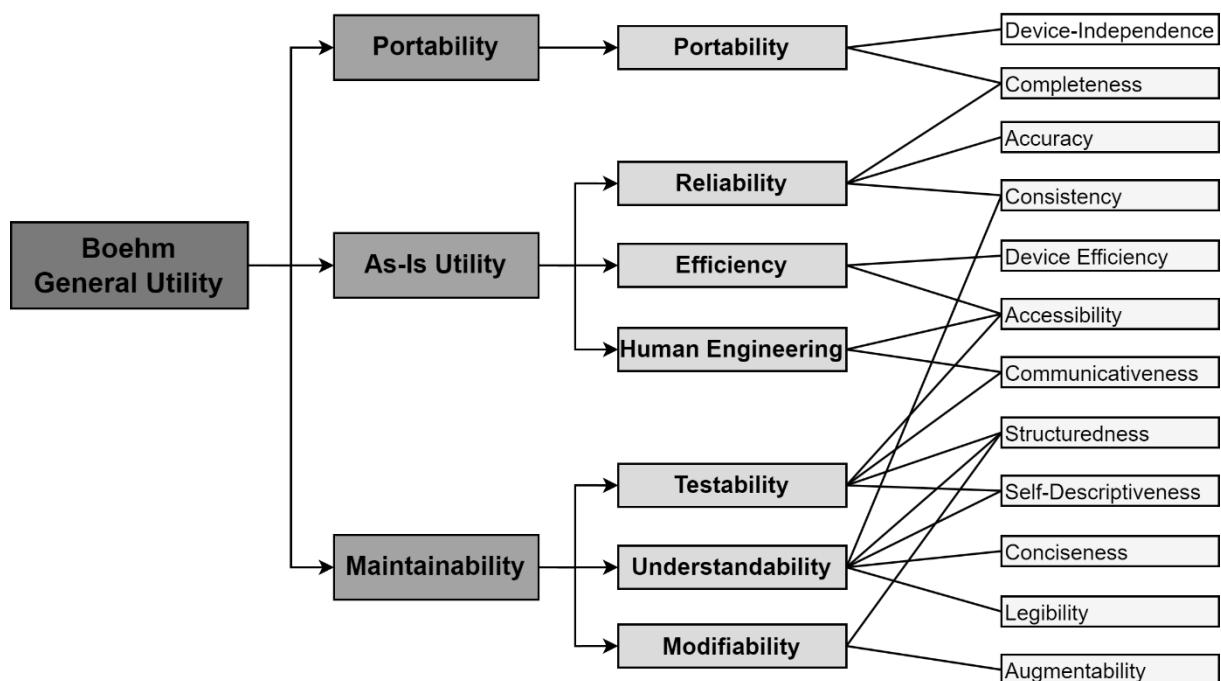


Abbildung 13: Qualitätseigenschaften nach Boehm [27] (eigene Abbildung)

2.4.2 McCall

Das Qualitätsmodell nach McCall [28] wurde im Jahre 1977 konzipiert und beschreibt ein dreistufiges Modell, welches aus 11 Qualitätshauptzielen besteht. Diese Hauptziele beinhalten die entsprechenden Qualitätsfaktoren ("factors"), die Qualitätskriterien ("criteria") für jeden Faktor und die dazugehörigen Metriken ("metrics").

Eine Metrik wird wiederum in binäre und relative Messungen eingeteilt. Während binäre Messungen meist eine Ja-Nein-Frage charakterisieren, beschäftigen sich relative Messungen mit Quotienten, also dem Verhältnis zweier Werte.

Nachfolgend werden zwei konkrete Fragen erläutert, um den Unterschied zwischen binären und relativen Messungen besser verstehen zu können:

- "*Ist das System intuitiv?*" stellte eine *binäre Messung der Benutzbarkeit* dar.
- "*Wie viele Stunden läuft die Software fehlerfrei in Relation zur Gesamtbetriebszeit?*" stellt eine *relative Messung der Zuverlässigkeit* dar.

Nun soll erläutert werden, welche Qualitätsfaktoren das Qualitätsmodell nach McCall umfasst und welche Qualitätskriterien mit welchen Qualitätsfaktoren verbunden sind. Diese Verknüpfungen sind essenziell, da sie die Qualitätsfaktoren nachvollziehbar beschreiben und messbar machen:

1. **Correctness** (Korrektheit)

bezieht sich darauf, inwiefern ein System die vorab definierten Anforderungen und Spezifikationen erfüllt. Qualitätskriterien für die Korrektheit sind Vollständigkeit, Verfolgbarkeit und Konsistenz.

2. **Reliability** (Zuverlässigkeit)

beschreibt die Leistungsfähigkeit eines Systems, die geforderten Funktionalitäten unter bestimmten Rahmenbedingungen zu erfüllen. Demnach wird also untersucht, ob das System dauerhaft und beständig ist. Qualitätskriterien für die Zuverlässigkeit sind Fehlertoleranz, Konsistenz und Genauigkeit.

3. **Efficiency** (Effizienz)

legt fest, inwieweit ein System seine Leistung mit minimalem Ressourcenverbrauch aufbringt. Qualitätskriterien für die Effizienz sind Ausführungs- und Speichereffizienz.

4. **Integrity** (Integrität)

zeigt auf, wie sicher ein System gegenüber unbefugtem Zugriff beziehungsweise unberechtigten Veränderungen auf Software und Daten ist. Qualitätskriterien für die Integrität sind Zugriffskontrolle und Zugriffsnachweis.

5. Usability (Benutzbarkeit)

erläutert, wie einfach es für einen Benutzer ist, das System kennenzulernen und sinnvoll verwenden zu können. Qualitätskriterien für die Integrität sind Kommunikativität, Trainierbarkeit und Bedienbarkeit.

6. Maintainability (Wartbarkeit)

bezeichnet den Aufwand, besser gesagt die Leichtigkeit, mit der ein System verändert werden muss, um Änderungen durchzuführen oder Fehler zu beheben. Qualitätskriterien für die Wartbarkeit sind Selbsterklärung, Kürze, Einfachheit und Modularität.

7. Testability (Testbarkeit)

ist definiert als der Aufwand, der entsteht, um Testfälle zu erstellen und Tests durchzuführen. Qualitätskriterien für die Testbarkeit sind Selbsterklärung, Instrumentierbarkeit, Einfachheit und Modularität.

8. Flexibility (Flexibilität)

gibt Aufschluss darüber, wie aufwändig beziehungsweise einfach ein System an neue Anforderungen angepasst werden kann. Qualitätskriterien für die Flexibilität sind Allgemeinheit, Erweiterbarkeit, Einfachheit und Modularität.

9. Portability (Portabilität)

hinterfragt, wie einfach ein System in andere Soft- oder Hardwareumgebungen übertragen werden kann. Qualitätskriterien für die Portabilität sind Einfachheit, Geräte und Systemunabhängigkeit.

10. Reusability (Wiederverwendbarkeit)

durchleuchtet, in welchem Ausmaß ein System oder dessen Komponenten erneut verwendet werden können. Qualitätskriterien für die Portabilität sind Allgemeinheit, Einfachheit, Modularität, Geräte und Systemunabhängigkeit.

11. Interoperability (Verknüpfbarkeit)

legt dar, wie unkompliziert ein System mit anderen Systemen verbunden werden kann, um Informationen auszutauschen und effektiv zusammenzuarbeiten. Qualitätskriterien für die Verknüpfbarkeit sind Datenkompatibilität, Kommunikationskompatibilität und Modularität.

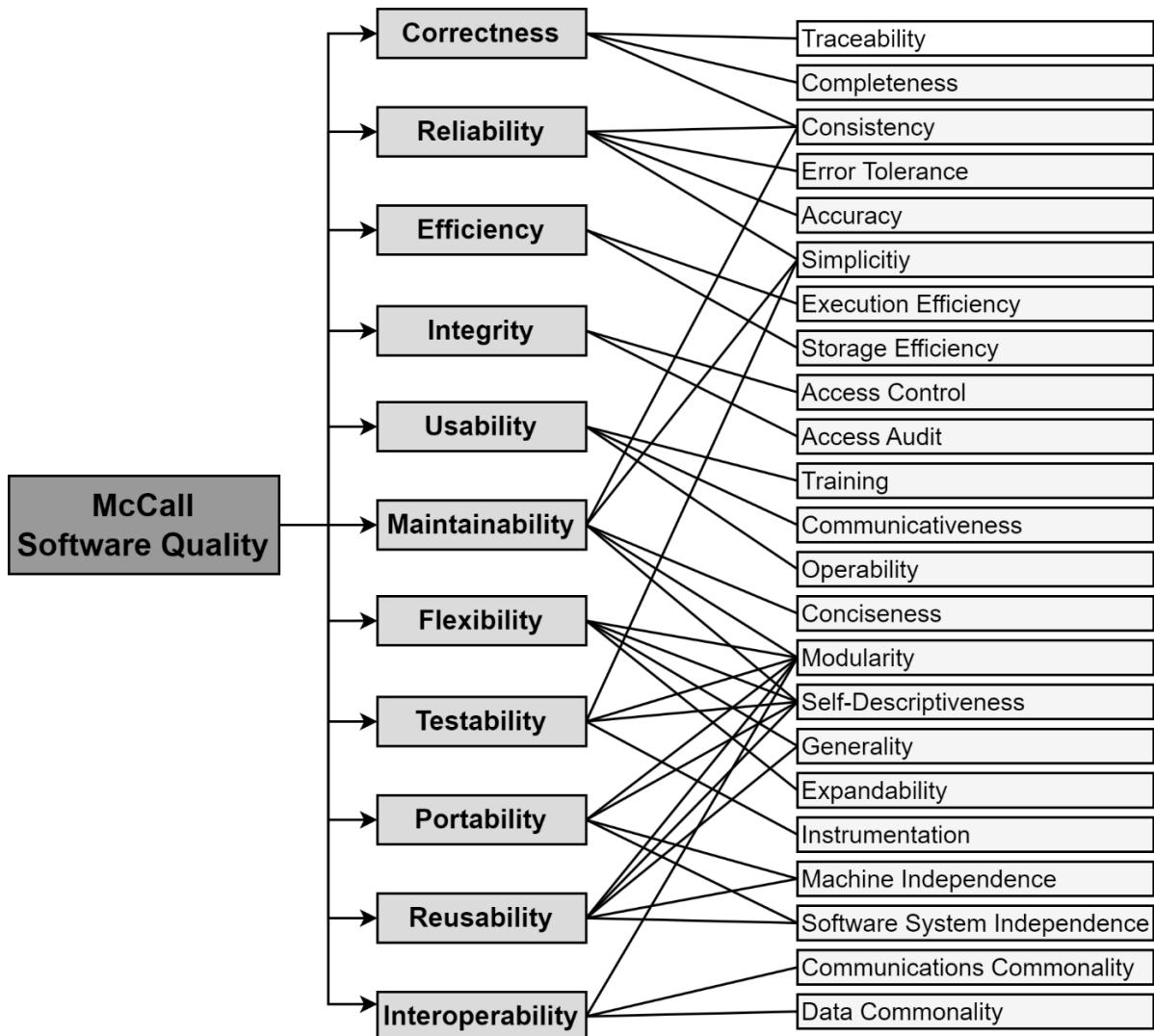


Abbildung 14: Qualitätseigenschaften nach McCall [28] (eigene Abbildung)

2.4.3 FURPS (+)

Das FURPS Modell ist dem Modell von McCall und dem von Boehm recht ähnlich, wobei die Qualitätseigenschaften in die Kategorien, *funktional* (F) und *nicht-funktional* (URPS), unterteilt werden. Funktionale Anforderungen beschreiben, welche spezifischen Aufgaben erledigt und welche Aktivitäten durchgeführt werden müssen. Nicht-funktionale Anforderungen definieren Kriterien, welche den Betrieb eines Systems bewerten. Das Modell wurde zwar ursprünglich vom amerikanischen Informationstechnologie Unternehmen Hewlett-Packard entwickelt, ist aber erst im Jahre 1992 von Robert Grady [29] veröffentlicht worden. Das Modell erhält sein Akronym durch folgende Qualitätsmerkmale:

- 1. Functionality (Funktionalität)**

wird durch den Funktionsumfang, den Systemfähigkeiten und der Systemsicherheit definiert.

2. **Usability** (Benutzbarkeit)

wird mithilfe der Benutzerfreundlichkeit, der Ästhetik, der Konsistenz, der Attraktivität und der Dokumentation festgelegt.

3. **Reliability** (Zuverlässigkeit)

wird durch die Fehlerhäufigkeit, die Wiederherstellbarkeit, die Genauigkeit und die gemittelte Zeit zwischen Ausfällen bestimmt.

4. **Performance** (Leistung)

wird anhand der Antwortgeschwindigkeit, der Effizienz, der Ressourcennutzung und des Durchsatzes gemessen.

5. **Supportability** (Unterstützbarkeit)

wird über die Testbarkeit, die Erweiterbarkeit, die Anpassungsfähigkeit, die Wartbarkeit, die Kompatibilität, die Konfigurierbarkeit, die Wartungsfähigkeit, die Installierbarkeit und die Lokalisierbarkeit ermittelt.

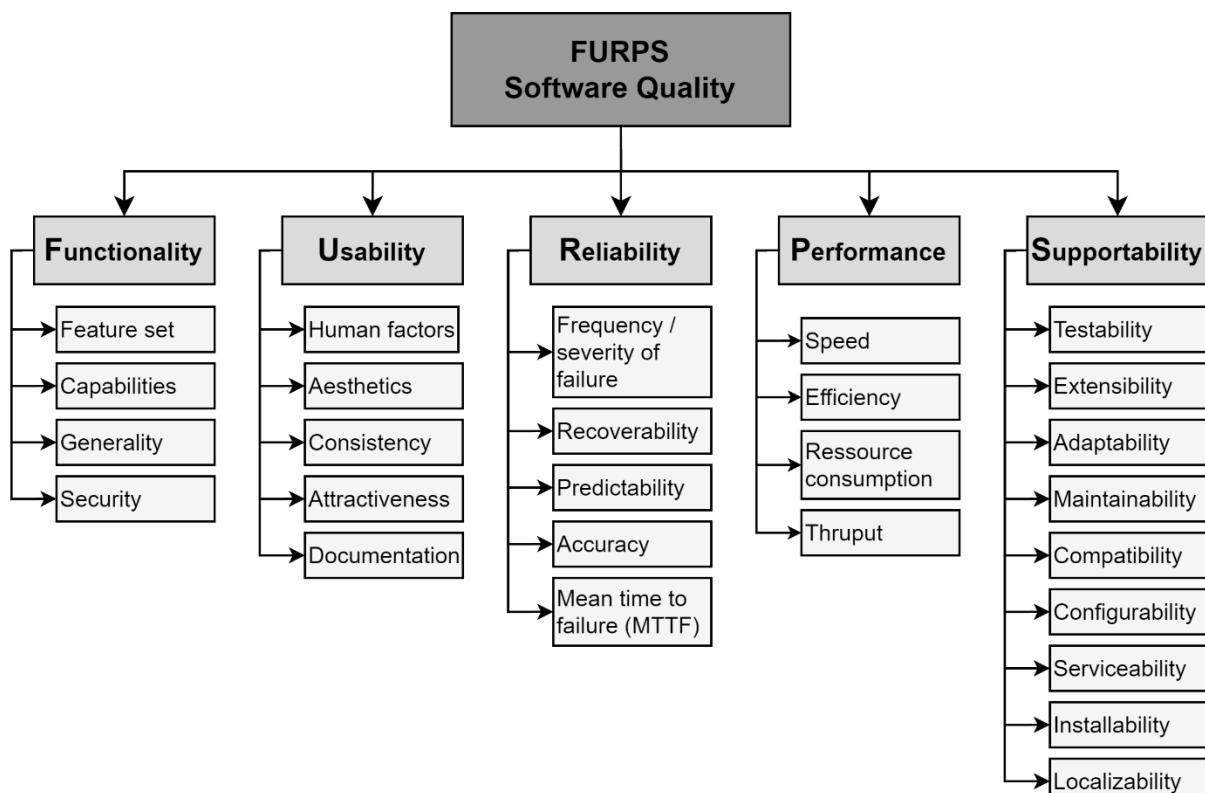


Abbildung 15: Qualitätseigenschaften nach FURPS [29] (eigene Abbildung)

Im Jahre 2000 wurde das FURPS-Modell von IBM Rational Software erweitert und zu „FURPS+“ umgetauft. Seit dem Zeitpunkt an, werden auch folgende Aspekte berücksichtigt:

- Designvorgaben
- Implementierungsanforderungen
- Schnittstellenanforderungen
- Physikalische Anforderungen

2.4.4 Goal Question Metric

Der Goal Question Metric (GQM) Ansatz wurde im Jahre 1994 von Victor Basili [30] erstellt und beschreibt eine umfassende Methode zur Entwicklung von Qualitätsmodellen für Softwaresysteme. Ursprünglich wurde die Metrik zur Evaluierung von Mängeln und Schäden innerhalb von Projekten des NASA Goddard Space Flight Centers genutzt. Obwohl sie anfänglich auf individuelle Projekte zugeschnitten wurde, wurde sie aufgrund der flexiblen Anwendbarkeit auf einen verallgemeinerten Kontext adaptiert. Der Ansatz basiert auf dem Gedankengang, dass eine zielgerichtete Messung nur erfolgen kann, wenn eine Organisation vorab klare Ziele für sich selbst und ihre Projekte gesetzt hat. Damit diese Ziele operativ beschrieben werden können, ist eine Zuweisung der Ziele zu den jeweiligen relevanten Informationsbedürfnissen essenziell. Um letztlich einen ordnungsgemäßen Interpretationsrahmen der zuvor definierten Ziele gewährleisten zu können, beispielsweise um zu analysieren, ob Ziele erreicht werden oder nicht, sind die zu quantifizierenden Informationsbedürfnisse der Organisation klarzustellen. Die Metrik besteht grundlegend aus drei Ebenen:

1. **GOAL** Konzeptionelle Ebene

Das Ziel bezieht sich immer auf eines der folgenden Objekte:

- **Products** (Produkte)

Produkte sind Ergebnisse, Artefakte oder Dokumente, die während des Lebenszyklus eines Systems erstellt werden. (z.B.: Entwürfe, Spezifikationen, Programme)

- **Processes** (Prozesse)

Prozesse umfassen jene Aktivitäten, welche sich auf die Software beziehen und meist zeitlich gebunden sind. (z.B.: Spezifizieren, Entwerfen, Testen, Befragen)

- **Resources** (Ressourcen)

Ressourcen beschreiben Elemente, die vom Prozess genutzt werden, um die dementsprechenden Ergebnisse erzeugen zu können. (z.B.: Hardware, Software, Büroräume, Personal)

2. **QUESTION** - Operationale Ebene

Hier werden mehrere Fragen formuliert, welche zur Erreichung bzw. Bewertung des zuvor definierten Ziels notwendig sind.

3. **METRIC** - Quantitative Ebene

Mehrere Metriken werden verwendet, um die erstellten Fragen in einer messbaren Weise beantworten zu können. Dabei werden folgende Arten von Metriken unterschieden:

- Objektiv
- Subjektiv

Grafisch dargestellt sieht das GQM-Modell folgendermaßen aus:

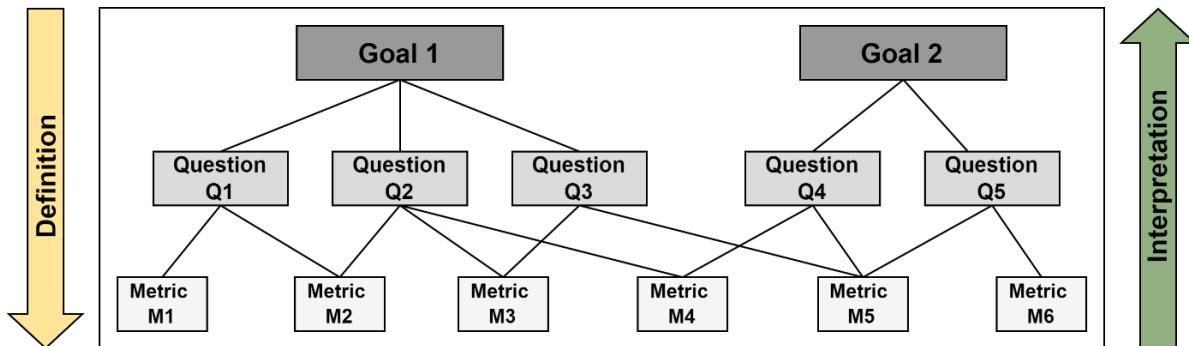


Abbildung 16: Systematische Darstellung des Goal Question Metric-Modells [30] (eigene Abbildung)

Wie Abbildung 16 bereits offenkundig zeigt, handelt es sich beim GQM-Modell um ein hierarchisches Top-Down Modell, welches mit der Definition des Ziels startet. Das Ziel beinhaltet:

- Den **Zweck** der Messung.
- Das zu messende **Problem**
- Das zu messende **Objekt** bzw. der zu messende **Prozess**.
- Den **Blickwinkel**, aus dem die Messung vorgenommen wird.

Anhand des jeweiligen Ziels werden dann wiederum Fragen abgeleitet, welche das Problem in seine Hauptkomponenten zerlegt. Jede dieser Fragen wird letztlich durch eine quantitative, also messbare, Metrik präzisiert, wobei die Metrik entweder subjektiv oder objektiv sein kann. Wie es auch teilweise in der obigen Abbildung der Fall ist, kann eine gleiche Metrik mehreren, unterschiedlichen Fragen zugewiesen werden. In manchen Sachverhalten kann es auch hilfreich sein, dieselbe Frage und dieselbe Metrik aus verschiedenen Blickwinkeln zu betrachten und deshalb mehrere GQM-Modelle zu erstellen.

Um den Einsatz des Goal Question Metric-Ansatzes besser nachvollziehen zu können, wird folgendes Beispiel verwendet: Ein IT-Unternehmen möchte die Pünktlichkeit der Fehlerbehebung während der Wartungsphase eines Systems verbessern. Das daraus entstehende Ziel definiert einen Zweck (engl. "Purpose"), ein Qualitätsproblem (engl. "Issue"), einen Prozess (engl. "Process") und einen Blickwinkel (engl. "Viewpoint"). Das für diesen Sachverhalt entstandene Goal Question Metric Modell kann folgendermaßen aussehen:

Tabelle 3: Beispielhafte Anwendung des GQM-Ansatzes [30]

Goal	Purpose Issue Object (process) Viewpoint	Improve the timeliness of the bug-fixing process from the project manager's viewpoint
Question		What is the current bug fixing process?
Metrics		<ul style="list-style-type: none"> Average cycle time

	<ul style="list-style-type: none"> • Standard deviation • Percentage of bugs exceeding the upper time limit
Question	Is the performance of the process improving?
Metrics	<ul style="list-style-type: none"> • $(\text{Current avg. cycle time} / \text{Baseline avg. cycle time}) * 100$ • Subjective rating of project manager's satisfaction

Es zeigt sich, dass das Ziel optimalerweise nur einen Satz umfasst und überhaupt nicht darauf eingehet, wie das Ziel erreicht werden soll. Ob ein Ziel korrekt formuliert wurde, lässt sich daran bemessen, ob eine Aufteilung in die Kategorien Zweck, Qualitätsproblem, Prozess und Blickwinkel erfolgen kann. Sollte es zudem der Fall sein, dass mehrere Ziele innerhalb eines Satzes definiert werden, ist es ratsam, diese aufzuteilen und getrennt voneinander zu behandeln bzw. zu betrachten.

2.4.5 Dromey

Das Dromey Qualitätsmodell [31] wurde von J. Dromey im Jahr 1995 entwickelt und beschreibt ein produktbasiertes Bottom-Up Model. Der Grundgedanke hinter dem Modell ist, dass eine Qualitätsmessung für jedes Produkt bzw. Systemkomponente unterschiedlich zu erfolgen hat.

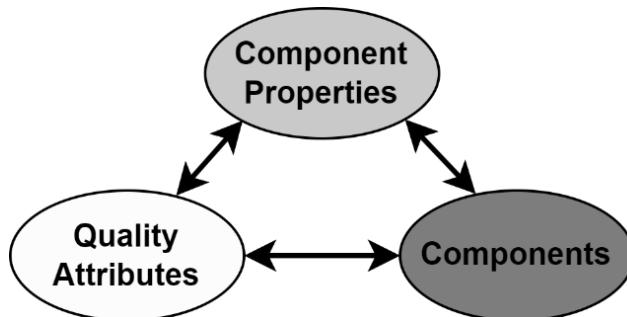


Abbildung 17: Grundgedanke des Dromey Qualitätsmodells [31] (eigene Abbildung)

In seinem Modell wird mit der Definition von Systemelementen (engl. „Components“), beispielweise Klassen oder Variablen, begonnen. Diese Komponenten werden mit qualitätstragenden Eigenschaften (engl. „Quality-carrying properties“) verbunden, welche in die Kategorien Korrektheit (engl. „Correctness“), Intern (engl. „Internal“), Kontextbezogenheit (engl. „Contextual“) und Beschreibungsqualität (engl. „Descriptive“) klassifiziert werden. Jede dieser Kategorien ist wiederum einem Hauptqualitätsmerkmal der ISO 9126:1991 zugeordnet, wobei das Merkmal Wiederverwendbarkeit (engl. „Reusability“) ergänzt wurde.

Das Vorgehen lautet nun wie folgt:

1. Definition bzw. Auswahl des Systemelements
2. Ablesen der dazugehörigen qualitätstragenden Eigenschaften (siehe Tabelle 4)

Tabelle 4: Beziehung zwischen Hauptqualitätsmerkmalen und qualitätstragenden Eigenschaften [31]

	Corrective Properties	Structural Properties				Modularity Properties			Descriptive Properties																	
	Computable	Complete	Assigned	Precise	Initialized	Progressive	Variant	Consistent	Structured	Resolved	Homogeneous	Effective	Nonredundant	Direct	Adjustable	Range-independent	Utilized	Parametrized	Loosely coupled	Encapsulated	Cohesive	Generic	Abstract	Specified	Documented	Self-descriptive
Object																										
Module																										
Sequence																										
Loop																										
Selection																										
Module calls																										
Assignment																										
Guard																										
Expression																										
Records																										
Variables																										
Constants																										
Types																										

3. Ablesen der beeinflussten Hauptqualitätsmerkmale (siehe Tabelle 5)

Tabelle 5: Beziehung zwischen Hauptqualitätsmerkmalen und qualitätstragenden Eigenschaften [31]

	Corrective Properties				Structural Properties				Modularity Properties				Descriptive Properties													
	Computable	Complete	Assigned	Precise	Initialized	Progressive	Variant	Consistent	Structured	Resolved	Homogeneous	Effective	Nonredundant	Direct	Adjustable	Range-independent	Utilized	Parametrized	Loosely coupled	Encapsulated	Cohesive	Generic	Abstract	Specified	Documented	Self-descriptive
Functionality																										
Reliability																										
Usability																										
Efficiency																										
Maintainability																										
Portability																										
Reusability																										

Um die Anwendung des Dromey Qualitätsmodells besser nachvollziehen zu können, ist folgendes Beispiel hilfreich, in welchem eine Variable analysiert wird:

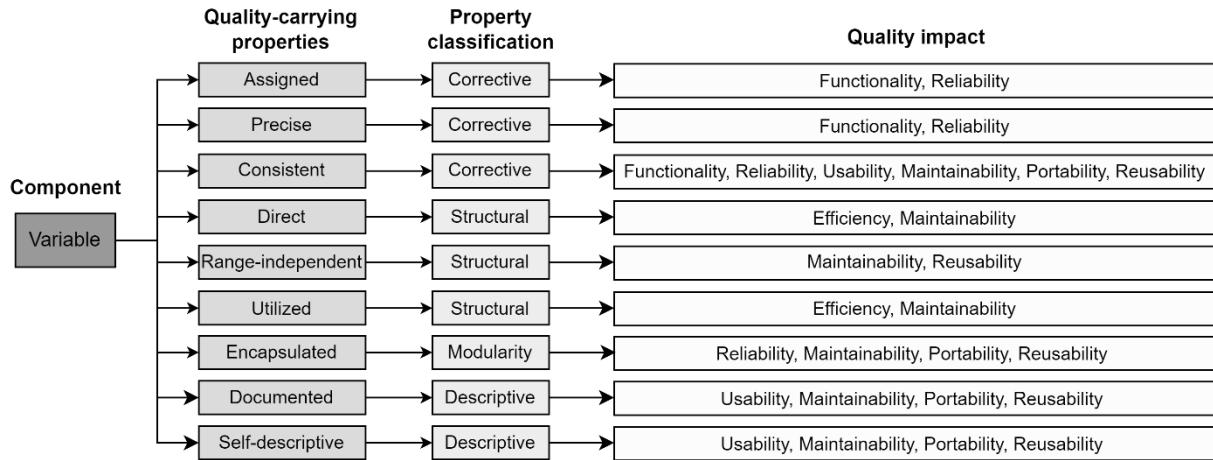


Abbildung 18: Beispielhafte Anwendung des Dromey Qualitätsmodells [31] (eigene Abbildung)

Zur besseren Nachvollziehbarkeit sollte die detaillierte Auflistung der Definitionen sämtlicher qualitätstragender Merkmale unter [31] nachgelesen werden.

Insbesondere, wenn das objektorientierte Design relevant ist, sollte auch das auf dem Dromey Modell basierende QMOOD-Modell (Quality Model for Object-Oriented Design) [32] in Anbetracht gezogen werden.

2.4.6 ISO/IEC 9126

Die Norm ISO/IEC 9126:1991 [33], welche ursprünglich im Jahre 1991 erschien, wurde im Jahre 2001 in vier Teile aufgeteilt und erhielt eine genauere Deklarierung der damaligen Hauptqualitätsmerkmale durch eine Zuweisung der dazugehörigen Teilmerkmale. Der erste Teil der Norm ISO/IEC 9126:2001 [34], um welchen es sich hier handelt, definiert ein beschreibendes Qualitätsmodell, welches Qualitätseigenschaften von Software-Systemen klassifiziert, jedoch nicht quantifiziert. Grundsätzlich werden sechs Merkmale definiert:

1. **Functionality** (Funktionalität)
Wurden die geforderten Funktionen implementiert und funktionieren sie auch?
2. **Reliability** (Zuverlässigkeit)
Kann eine angemessene Fehlerfreiheit gewährleistet werden?
3. **Usability** (Benutzbarkeit)
Wie hoch ist der Aufwand für einen Benutzer, um das System effektiv nutzen zu können?
4. **Efficiency** (Effizienz)
Wie effizient ist die Applikation, sprich das Verhältnis zwischen der Systemleistung und den eingesetzten Betriebsmitteln?
5. **Maintainability** (Wartbarkeit)
Wie einfach ist es, das System an neue Anforderungen oder Umgebungen anzupassen?

6. Portability (Portabilität)

Kann das System in eine andere Soft- oder Hardwareumgebung migriert werden?

Diesen Merkmalen werden wiederum jeweils drei bis sechs Teilmerkmale zugeordnet, welche die Beschreibung weiter konkretisieren. Anders als das Qualitätsmodell nach McCall, handelt es sich hierbei um ein streng hierarchisch angeordnetes Modell. Sinnbildlich kann die Norm in folgender Abbildung zusammengefasst werden:

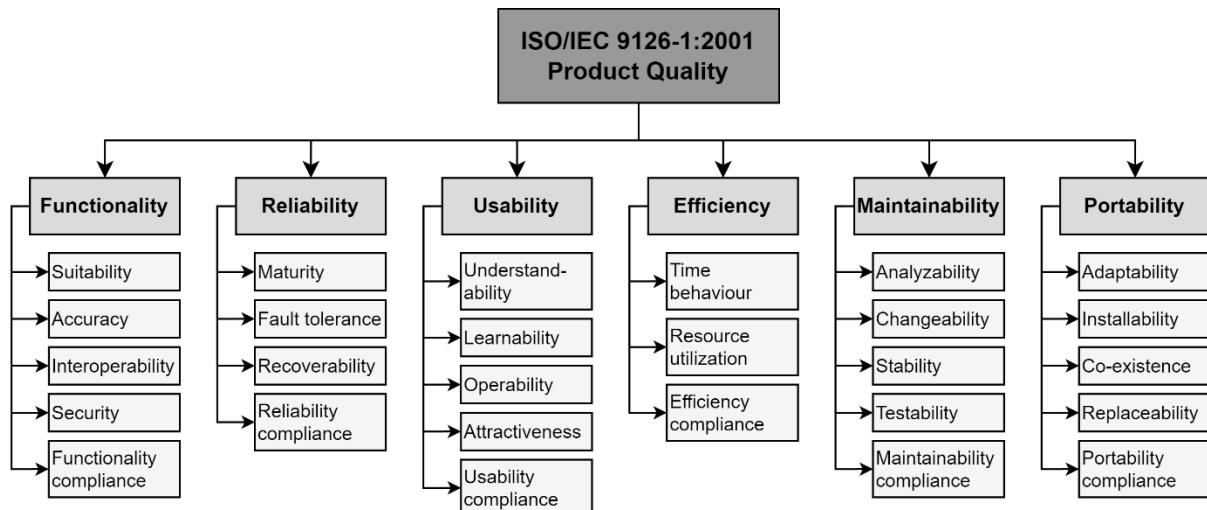


Abbildung 19: Qualitätseigenschaften von Softwaresystemen nach ISO/IEC 9126-1:2001 [33] (eigene Abbildung)

In Summe beinhaltet die Norm ISO/IEC 9126-1:2001 also 27 beschreibende Qualitätsmerkmale.

2.4.7 ISO/IEC 25010

ISO/IEC 25010:2011

Im Jahre 2011 wurde die Norm ISO/IEC 9126-1:2001 durch die Norm ISO/IEC 25010:2011 [35] ersetzt. Es handelt sich dabei ebenfalls um ein beschreibendes Qualitätsmodell, welches die Qualitätseigenschaften in acht (anstelle von sechs) Merkmale unterteilt:

1. **Functional Suitability** (Funktionale Eignung)
Wurden die geforderten Funktionen implementiert und funktionieren sie auch?
2. **Performance Efficiency** (Leistungseffizienz)
Kann eine angemessene Fehlerfreiheit gewährleistet werden?
3. **Compatibility** (Kompatibilität)
Wie hoch ist der Aufwand für einen Benutzer, um das System effektiv nutzen zu können?
4. **Usability** (Benutzbarkeit)
Wie effizient ist die Applikation, sprich das Verhältnis zwischen der Systemleistung und den eingesetzten Betriebsmitteln?

5. Reliability (Zuverlässigkeit)

Wie einfach ist es, das System an neue Anforderungen oder Umgebungen anzupassen?

6. Security (Sicherheit)

Kann das System in eine andere Soft- oder Hardwareumgebung migriert werden?

7. Maintainability (Wartbarkeit)

Wird das System vor unbefugtem Zugriff geschützt und kann ein Schutz von Daten und Informationen gewährleistet werden?

8. Portability (Portabilität)

Inwieweit funktioniert das System in Kombination mit anderen Soft- und Hardwarelösungen?

Wie auch schon bei der Norm ISO/IEC 9126-1:2001 gibt es den Merkmalen untergeordnete Teilmerkmale, wodurch sich ebenfalls ein streng hierarchisch angeordnetes Modell bildet. Die erläuterten Qualitätseigenschaften werden hier ebenfalls nicht quantifiziert. Grafisch dargestellt ergibt sich daraus folgende Abbildung:

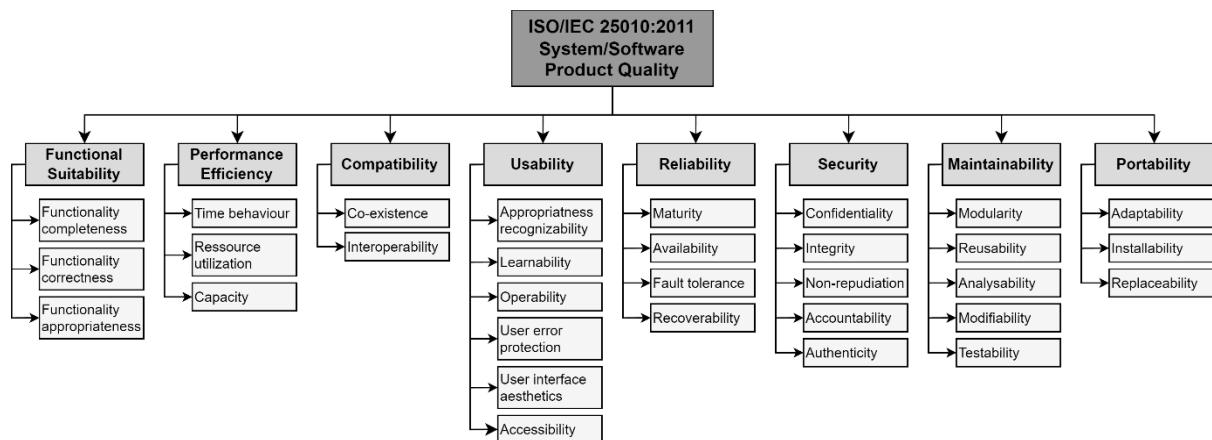


Abbildung 20: Qualitätseigenschaften von Softwaresystemen nach ISO/IEC 25010:2011 [35]
(eigene Abbildung)

Folgende Änderungen können im Vergleich zur ISO/IEC 9126:2001 beobachtet werden:

1. Einführung von neuen Hauptmerkmalen:
 - *Security*
 - *Compatibility*
2. Umbenennung von Hauptmerkmalen:
 - *Functionality* → *Functional Suitability*
 - *Efficiency* → *Performance Efficiency*
3. Einführung von neuen Teilmerkmalen:
 - *Availability* (unter Reliability)
 - *Capacity* (unter Performance Efficiency)
 - *Interoperability* (unter Compatibility)

- *Confidentiality* (unter Security)
- *Integrity* (unter Security)
- *Non-repudiation* (unter Security)
- *Accountability* (unter Security)
- *Authenticity* (unter Security)

4. Verschiebung von Teilmerkmalen:

- *Co-existence* (vorher unter Portability, jetzt unter Compatibility)
- *Interoperability* (vorher unter Functionality, jetzt unter Compatibility)

5. Entfernung von alten Teilmerkmalen:

- *Functionality compliance*
- *Reliability compliance*
- *Usability compliance*
- *Efficiency compliance*
- *Maintainability compliance*
- *Portability compliance*

6. Genauere Spezifizierung von Teilmerkmalen:

- | | |
|---------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> • <i>Suitability</i> • <i>Accuracy</i> | <ul style="list-style-type: none"> • <i>Functionality completeness</i> • <i>Functionality correctness</i> • <i>Functionality appropriateness</i> |
| <ul style="list-style-type: none"> • <i>Understandability</i> • <i>Attractiveness</i> | <ul style="list-style-type: none"> • <i>Appropriateness recognizability</i> • <i>User error protection</i> • <i>User interface aesthetics</i> • <i>Accessibility</i> |
| <ul style="list-style-type: none"> • <i>Changeability</i> • <i>Stability</i> | <ul style="list-style-type: none"> • <i>Modularity</i> • <i>Reusability</i> • <i>Modifiability</i> |

Hier sei jedoch angemerkt, dass einige Änderungen schwer nachvollziehbar sind, und möglicherweise zu Fehlinterpretation führen könnten. Die fundamentale Umstrukturierung bzw. Erweiterung zeigt jedenfalls auf, dass die Erstellung und/oder Optimierung eines Qualitätsmodells sehr anspruchsvoll und aufwendig ist.

Summa Summarum beinhaltet die Norm ISO/IEC 25010:2011 31 beschreibende Qualitätsmerkmale.

ISO/IEC 25010:2023

Im Jahre 2023 wurde die Norm ISO/IEC 25010:2011 zurückgezogen und durch die Norm ISO/IEC 25010:2023 [36] ersetzt:

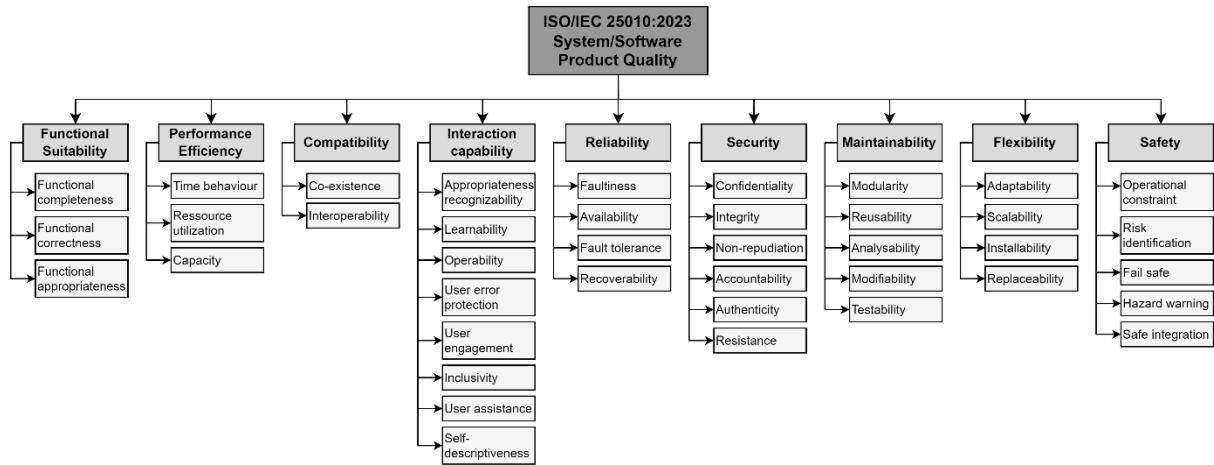


Abbildung 21: Qualitätseigenschaften von Softwaresystemen nach ISO/IEC 25010:2023 [36]
(eigene Abbildung)

Folgende Änderungen können im Vergleich zur ISO/IEC 25010:2011 beobachtet werden:

1. Einführung von neuem Hauptmerkmal:
 - *Safety*
2. Umbenennung von Hauptmerkmalen:
 - *Usability* → *Interaction capability*
 - *Portability* → *Flexibility*
3. Einführung von neuen Teilmerkmalen:
 - *Operational constraint* (unter *Safety*)
 - *Risk identification* (unter *Safety*)
 - *Fail safe* (unter *Safety*)
 - *Hazard warning* (unter *Safety*)
 - *Safe integration* (unter *Safety*)
 - *Resistance* (unter *Security*)
 - *Scalability* (unter *Flexibility*)
4. Genaue Spezifizierung von Teilmerkmalen:
 - *User interface aesthetics* →
 - *User engagement*
 - *Inclusivity*
 - *User assistance*
 - *Self-descriptiveness*
 - *Accessibility*

Die Norm ISO/IEC 25010:2023 definiert somit 39 beschreibende Qualitätsmerkmale.

2.4.8 ISO/IEC 25059

Neben dem allgemeinen Qualitätsmodell für Software-Systeme, der Norm ISO/IEC 25010:2023, wurde im Jahre 2023 auch ein Qualitätsmodell spezifisch für Software-Systeme mit Fokus auf Künstlicher Intelligenz, veröffentlicht. Die Norm ISO/IEC 25059:2023 [37] ist eine

modifizierte Version der Norm ISO/IEC 25010:2011 und kann anhand der folgenden Abbildung besser nachvollzogen werden:

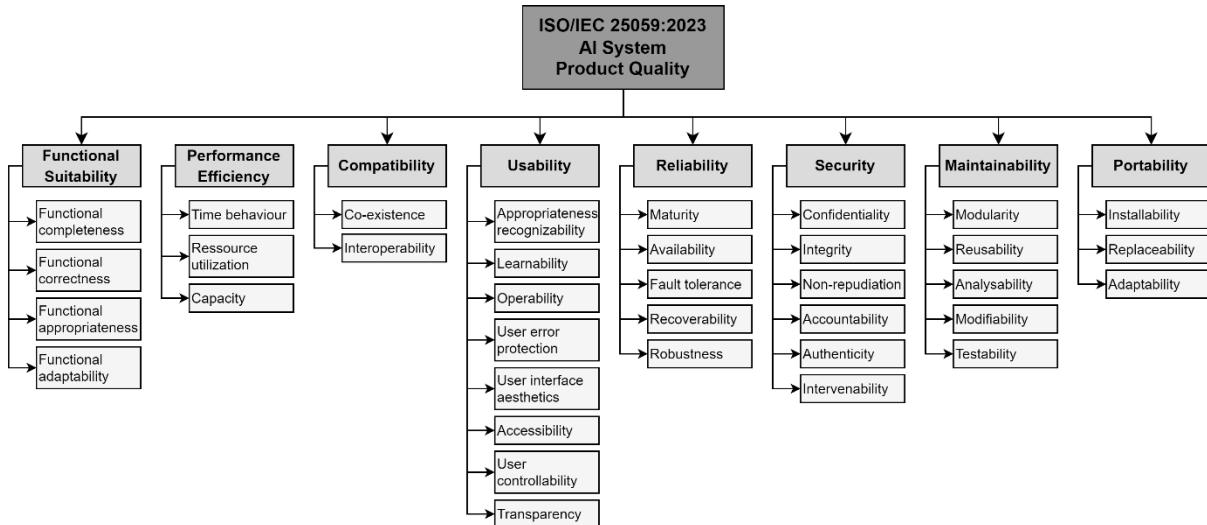


Abbildung 22: Qualitätseigenschaften von AI-Softwaresystemen nach ISO/IEC 25059:2023 [37]
(eigene Abbildung)

Folgende Änderungen können im Vergleich zur ISO/IEC 25010:2011 beobachtet werden:

1. Einführung von neuen Teilmerkmalen:

- *Functional adaptability* (unter Functional Suitability)
- *User controllability* (unter Usability)
- *Transparency* (unter Usability)
- *Robustness* (unter Reliability)
- *Intervenability* (unter Security)

Zusammengefasst definiert die Norm ISO/IEC 25059:2023 36 beschreibende Qualitätsmerkmale.

2.4.9 Arc 42 Q42

Dass es dem aktuellen Qualitätsstandard ISO/IEC 25010:2023 an praktischer Anwendbarkeit und Umsetzbarkeit mangelt, sehen auch die Ersteller des arc42 Qualitätsmodells [38] so. Das Modell ist auch unter dem Synonym Q42 (ausgesprochen „Kju-Fortytwo“ oder „Kju-Four-Two“) bekannt und wurde im Januar 2023 entwickelt. Es soll einen einfachen, aber auch effektiven Ansatz zur Bewertung von Produkt- und Systemqualität ermöglichen. Beginnend mit den Erwartungen und Anforderungen der Stakeholder, werden acht zentrale Systemeigenschaften definiert, die ausreichen, um die nötigen Aspekte aus über 158 traditionellen Qualitätsmerkmalen abzudecken. Ähnlich wie beim McCall-Modell wird der Gedankengang verfolgt, dass ein Qualitätsmerkmal (z.B.: Verfügbarkeit) mehrere Eigenschaften (z.B.: Zuverlässigkeit und Benutzbarkeit) erfüllen kann. Die acht zentralen Top-Level Systemeigenschaften sehen wie folgt aus:

1. **Reliable** (enthält 44 zugewiesene Qualitätsmerkmale)
Erfüllt die Software die definierten Funktionen unter bestimmten Bedingungen ohne Unterbrechungen und Probleme?
2. **Flexible** (enthält 31 zugewiesene Qualitätsmerkmale)
Wie einfach kann das Produkt an neue bzw. überarbeitete Anforderungen oder Systemumgebungen angepasst werden?
3. **Efficient** (enthält 37 zugewiesene Qualitätsmerkmale)
Erfüllt die Software seine Funktionen innerhalb der spezifizierten Zeit und weist sie dabei eine effiziente Ressourcennutzung auf (z.B.: Speicher, Threads, Bandbreite)?
4. **Usable** (enthält 61 zugewiesene Qualitätsmerkmale)
Wird den Nutzern ein positives Nutzungserlebnis ermöglicht und können sie ihre Aufgaben sicher, effektiv und effizient lösen?
5. **Safe** (enthält 10 zugewiesene Qualitätsmerkmale)
Werden Zustände, die dem Menschen oder der Umwelt schaden könnten, frühzeitig erkannt und vermieden?
6. **Secure** (enthält 19 zugewiesene Qualitätsmerkmale)
Werden Daten und Information so geschützt, dass unbefugte Personen nur begrenzten Zugriff haben?
7. **Suitable** (enthält 24 zugewiesene Qualitätsmerkmale)
Stellt die Software Eigenschaften bereit, die die Bedürfnisse der Stakeholder erfüllen?
8. **Operable** (enthält 34 zugewiesene Qualitätsmerkmale)
Ist die Software einfach zu installieren, zu überwachen und zu administrieren?

Dem ist hinzuzufügen, dass zu den acht Systemeigenschaften, die jeweilig zu erwartenden Anforderungen der Stakeholder zugeordnet werden. Darin werden folgende Rollen berücksichtigt:

- Die Benutzer
- Die Produkt-Owner
- Das Management
- Die Entwickler
- Die Tester
- Die Administratoren
- Die Domain-Experten

Dies erinnert sehr an den GQM-Ansatz, welcher ebenfalls die Ideologie verfolgt, die verschiedenen Standpunkte bzw. die subjektiven Messungen der Stakeholder zu berücksichtigen.

2.4.10 Praxiseinsatz

Da im Rahmen dieser Arbeit keine Kapazität für Experteninterviews gegeben ist, soll auf bereits bestehende Umfragen zurückgegriffen werden. Die zum Zeitpunkt der Arbeit aktuellste Umfrage wurde im Jahre 2012 im Rahmen des Forschungsprojekts Quamoco vom deutschen Bundesministerium für Bildung und Forschung durchgeführt. An der Umfrage nahmen 125 Teilnehmer aus 12 Ländern teil. Von den Befragten hatten 29% 11 bis 15 Jahre und nur 15% weniger als fünf Jahre Arbeitserfahrung im Bereich der Softwareentwicklung. [39]

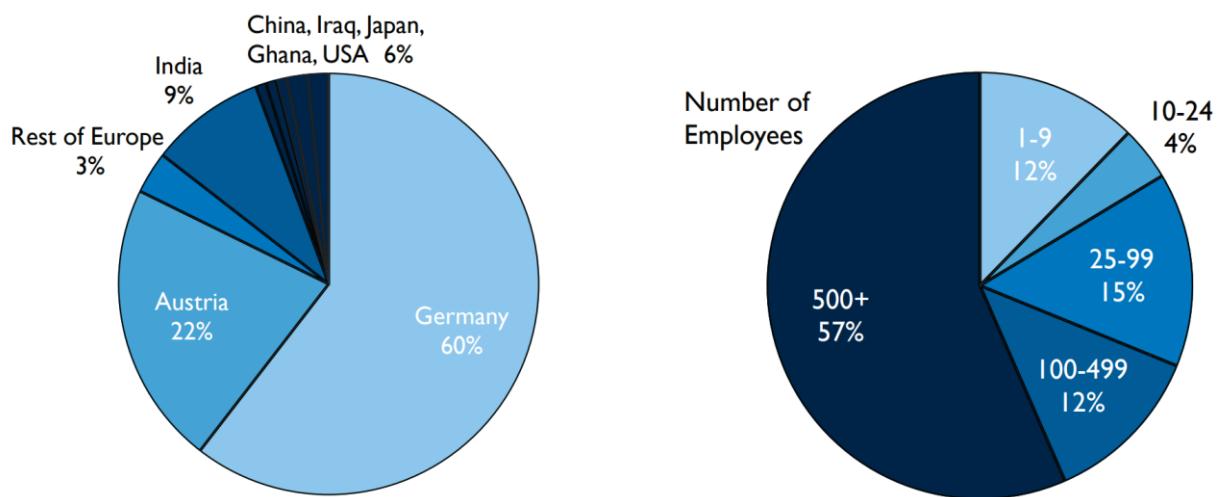


Abbildung 23: Herkunft und Unternehmensgröße der Befragten [39]

Welche Qualitätsmodelle werden verwendet, um Produktqualität sicherzustellen?

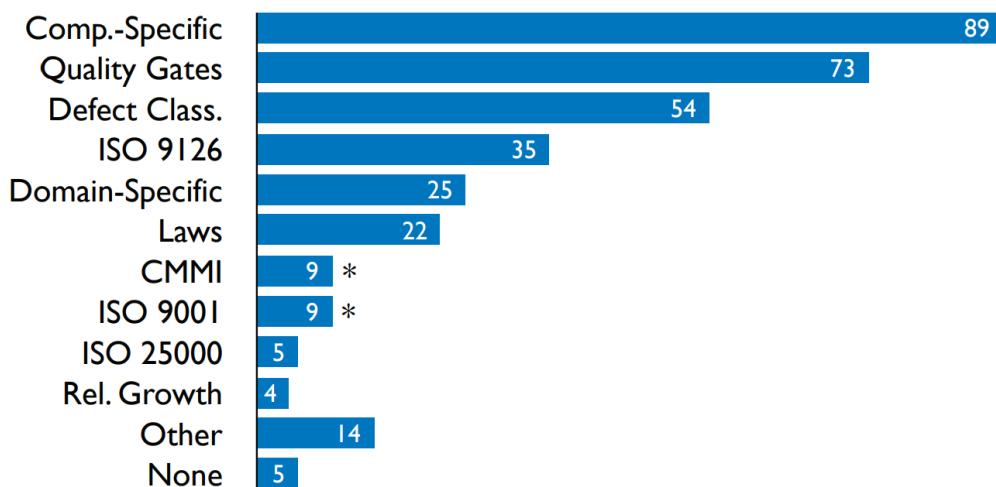


Abbildung 24: Praxiseinsatz von Qualitätsmodellen [39]

Es zeigt sich, dass in der Praxis vorrangig unternehmensspezifische Qualitätsmodelle verwendet werden. ISO-Standard-Modelle werden weniger akzeptiert: Von den Befragten nutzten lediglich 28% die ISO 9126 und 4% die ISO 25000. Davon haben 71,4% angegeben, neben den ISO-Standards auch unternehmensspezifische Modelle zu verwenden. Dies deutet

darauf hin, dass eine direkte Verwendung der ISO-Standard-Modelle die Bedürfnisse der Qualitätsexperten nicht erfüllt. In Anbetracht des Faktes, dass der Standard rund um die ISO 25000, welcher auch die ISO 25010 inkludiert, zur Zeit der Befragung frisch veröffentlicht wurde und noch teilweise unvollständig war, könnte dies eine Ursache für die mangelhafte Verbreitung sein. Eine Interpretation der damaligen Relevanz der ISO 9126 ist daher vermutlich aussagekräftiger.

Werden die Qualitätsmodelle individuell auf jedes Produkt angepasst?

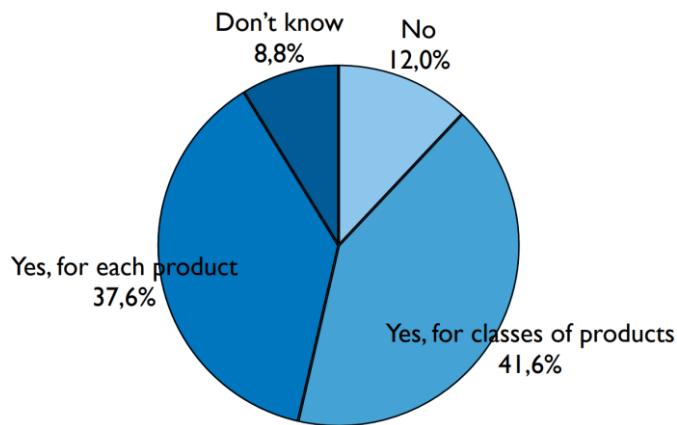


Abbildung 25: Anpassung der Qualitätsmodelle an individuelle Anforderungen [39]

Wichtig zu beachten ist, dass lediglich 12% der Befragten ihre Qualitätsmodelle nicht individuell anpassen. Die Mehrheit passt ihre Modelle entweder für einzelne Produkte (37,6%) oder für unterschiedliche Produktklassen (41,6%) an.

Wie häufig werden folgenden Techniken zur Qualitätsevaluation benutzt?

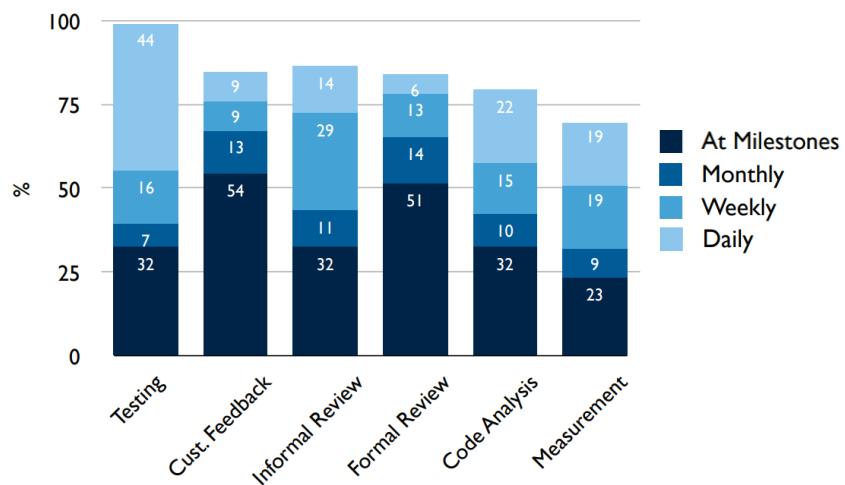


Abbildung 26: Nutzungs frequenz der verwendeten Qualitätssicherungsmethoden [39]

Angemerkt sei hier zuallererst, dass eine Eingabe von mehreren Antworten möglich war. Jene Qualitätssicherungsmethoden, welche automatisiert stattfinden können, wie es beispielsweise

beim Testen der Fall ist, werden meist täglich verwendet. Andere Methoden, welche einen höheren Personal- oder Ressourcenaufwand benötigen, wie zum Beispiel formelle Reviews, werden vorrangig bei Meilensteinen eingesetzt.

Welche drei Qualitätsmerkmale sollten, wenn möglich, verbessert werden?

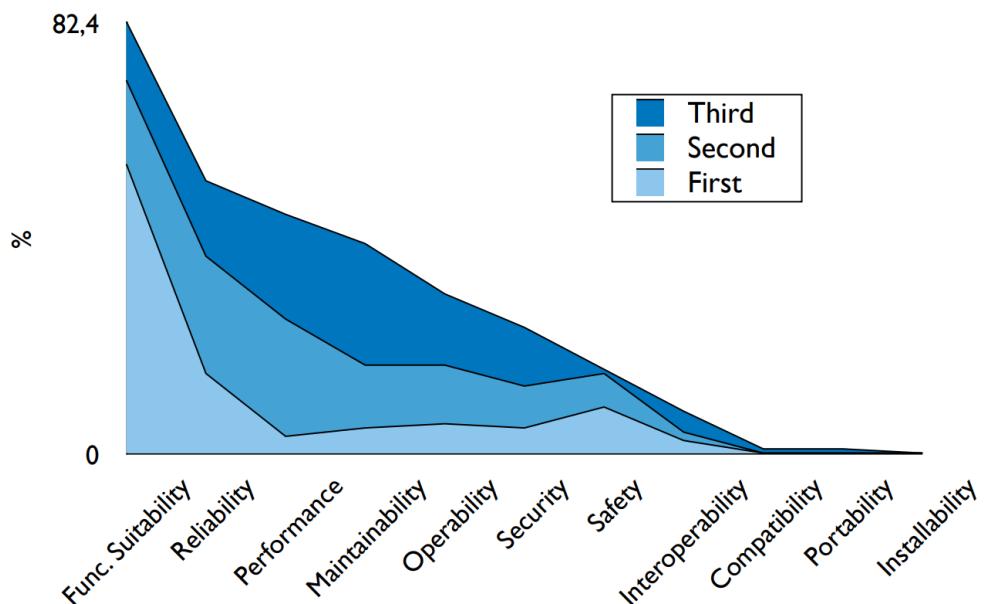


Abbildung 27: Die relevantesten Qualitätsmerkmale mit Verbesserungspotenzial [39]

Die Umfrage zeigte zudem, dass die Qualitätsmerkmale Functional Suitability, Reliability, Performance, Maintainability, Operability, Security und Safety am relevantesten sind. Diese Merkmale sollten deshalb, in welcher Form auch immer, in einem Qualitätsmodell enthalten sein. Im Kontrast dazu wurde Interoperability, Compatibility, Portability und Installability kaum bis gar nicht genannt.

2.4.11 Fazit

Um ein ordnungsgemäßes und aussagekräftiges Fazit ziehen zu können, ist eine Zusammenfassung der wichtigsten Modelle zur Qualitätsbewertung von Softwaresystemen besonders hilfreich:

Tabelle 6: Historie der relevantesten Software-Qualitätsmodelle

	Veröffentlichung	Struktur	Verknüpfung	Hauptmerkmale	Einzigartige Teilmerkmale
Boehm	1976	Top-Down	One to Many	3	15
McCall	1977	Top-Down	One to Many	11	23
ISO/IEC 9126:1991	1991	Top-Down	One to One	6	0
FURPS	1992	Top-Down	One to One	5	27
GQM	1994	Top-Down	individuell	individuell	individuell
Dromey	1995	Bottom-Up	One to One	6	2 bis 12
ISO/IEC 9126-1:2001	2001	Top-Down	One to One	6	27
ISO/IEC 25010:2011	2011	Top-Down	One to One	8	31
ISO/IEC 25059:2023	2023	Top-Down	One to One	8	36
ISO/IEC 25010:2023	2023	Top-Down	One to One	9	40
Arc42 Q42	2023	Top-Down	One to Many	8	158

Verlauf der ISO/IEC-Standards für Softwarequalität

Die Historie zeigt, dass die ISO/IEC-Standards für Softwarequalität mit jeder Neuerscheinung umfangreicher und komplexer wurden. Weiters kann festgehalten werden, dass die Anzahl der Haupt- und Teilmerkmale kontinuierlich zugenommen hat. Während die ursprüngliche Veröffentlichung der Norm ISO/IEC 9126:1999 noch überschaubare sechs Merkmale hatte, umfasst die Norm ISO/IEC 25010:2023 mittlerweile 39 beschreibende Teilmerkmale. Dies hängt womöglich mit den steigenden Anforderungen an Softwarequalität und mit der Notwendigkeit zusammen, diese auch tatsächlich präzise und umfassend bewerten zu können. Anbetracht des bisherigen Verlaufes der Ansprüche an ISO/IEC-Standard Qualitätsmodelle, lässt sich mutmaßen, dass künftig weitere Haupt- und/oder Teilmerkmale ergänzt werden könnten. Deswegen ist es äußerst ratsam die ISO-Website regelmäßig unter folgendem Link auf künftige Änderungen zu überprüfen:

<https://www.iso.org/standard/78176.html>

Darüber hinaus zeigt die ständige Adjustierung der Qualitätsmodelle, dass eine standardisierte Bewertung der Qualität von Softwaresystem schwierig zu realisieren ist. Einerseits weil jeder Entwickler, jeder Stakeholder, jedes Unternehmen und jede Industrie eine andere Vorstellung von spezifischen Qualitätseigenschaften hat und andererseits, weil die individuellen Anforderungen von Projekt zu Projekt maßgeblich variieren.

Optimierungsmöglichkeiten der ISO/IEC-Standards

Aus genannten Gründen ist es empfehlenswert, wie beim GQM-Ansatz, den Blickwinkel sowie die subjektive Wahrnehmung der Person oder des Teams, welches die Messung vornimmt, in die Bewertung miteinzubeziehen. Zwar bietet die Norm ISO/IEC 25010:2023 entsprechende Anwendungsbeispiele der jeweiligen Qualitätsmerkmale und damit einen informativen Kontext, doch eine genauere Spezifizierung der Bewertungskriterien bleibt aus. Auf Basis dieses Kontexts und der Anwendungsbeispiele sollte das Ziel des jeweiligen Qualitätsmerkmals konkretisiert werden. Danach können passende spezifische Fragen formuliert und abhängig davon subjektive als auch objektive Metriken ergänzt werden.

Es zeigt sich zudem, dass Modelle, die in einer Zeit erstellt wurden, als Software noch in den Kinderschuhen steckte, äußerst hilfreiche Ansätze aufweisen können.

- Der Gedanke der *flexiblen Vernetzung von mehreren Qualitätseigenschaften*, wie es beim Boehm- und McCall-Modell der Fall ist.
- Die weitere Einteilung der Metriken in *binäre* und *relative Messungen* auf Grundlage des McCall-Modells.
- Die Unterscheidung von *funktionalen* und *nicht-funktionalen* Anforderungen anhand des FURPS-Modells.
- Neben einer *Top-Down Struktur* ist auch der Einsatz einer *Bottom-Up Struktur*, wie beim Dromey-Modell, ein empfehlenswerter Ansatz. Beginnend mit den Systemkomponenten wie Variablen oder Klassen, wird schrittweise ein qualitativ hochwertiges Gesamtsystem aufgebaut.

Zusammengefasst kann die Kombination der aktuellen Norm, zur Zeit der Verfassung dieser Arbeit die ISO/IEC 25010:2023, mit der GQM-Methode, dem Boehm-, dem McCall- und eventuell dem Dromey-Modell ein optimiertes Softwarebewertungsverfahren ermöglichen. Soll jedoch kein individuelles erstellt, sondern auf ein bereits bestehendes Qualitätsmodell zurückgegriffen werden, so ist das Arc Q42 Qualitätsmodell empfehlenswert. Es punktet definitiv mit seiner ausführlichen Dokumentation, die durch klare Definitionen und praxisnahe Beispiele untermauert wird. Insbesondere für Anfänger oder Teams, die keine Kapazitäten haben, um sich mit komplexen Qualitätsmodellen, wie der ISO/IEC 25010:2023 oder dem Arc Q42 Modell auseinanderzusetzen, empfiehlt sich jedoch der Einsatz einfacherer Ansätze. Wichtig ist es jedenfalls, die Anforderungen sämtlicher Stakeholder zu berücksichtigen und das Modell abhängig vom jeweiligen Anwendungsfall individuell anzupassen.

2.5 Metriken für Softwaresysteme

Metriken werden in der Literatur von L. Lazic und N. Mastorakis wie folgt beschrieben:

“Metrics are measurements of the world around us. Without measurements, we are blind to the changes that go on in the world. Without measurements we can never know if we are improving or getting worse; we can never know if we are succeeding or failing.” [40] (S. 600)

Metriken sind also Werkzeuge bzw. Messungen, die Veränderungen sichtbar machen sollen, um Aussagen über den Erfolg oder Misserfolg eines Vorhabens treffen zu können. Sie tragen heutzutage einen wesentlichen Beitrag zur Bewertung des Erfolges von Softwaresystemen bei. Der Einsatz von Metriken wurde erstmals im Jahre 1968 von R. J. Rubey und R. D. Hartwick [41] diskutiert. Seither bilden sie einen fundamentalen Bestandteil der IT-Industrie und haben bereits folgende Erkenntnis geliefert: *“Simpler is almost always better”* [40]. Der Fokus sollte also nicht auf der Diskussion der Sinnhaftigkeit von Metriken liegen, sondern vielmehr darauf, welche Metriken zielführend sind und wie einfach diese in der Praxis angewendet werden können.

D. R. Andrews [42] hebt die folgenden drei grundlegende Funktionen von Metriken hervor:

- **Kontrolle:** Bewertung und Evaluierung des Projektfortschritts zur Sicherstellung der Zielerreichung.
- **Kommunikation:** Bereitstellung von Messergebnissen an interne und externe Stakeholder zur Förderung der Transparenz.
- **Verbesserung:** Identifikation und Analyse von Abweichungen zwischen Soll und Ist zur kontinuierlichen Prozessoptimierung.

M. Broy und M. Kuhrmann [43] definieren folgende Anforderungen an Metriken:

- Objektivität
- Aussagekraft und Tauglichkeit
- Nützlichkeit
- Vergleichbarkeit
- Messgenauigkeit
- Angemessenheit des Aufwands

Des Weiteren können Metriken laut M. Broy und M. Kuhrmann [43] wie folgt eingeteilt werden:

- **Produktmetriken** beschäftigen sich mit Umfang/Größe, Komplexität, Design, Performanz und Qualität des Produkts.
- **Prozessmetriken** messen den Zeitbedarf für Fehlerkorrekturen, den Zeitbedarf für die Einarbeitung von Änderungen und die Prozessreife (z.B.: CMMI).
- **Projektmetriken** erfassen Termintreue, Produktivität, Kosten- und Ressourcenverbrauch.

Im Rahmen dieser Arbeit liegt der Fokus auf Produktmetriken, weshalb Radon [44][41], ein statisches Analysetool in Python, zur Berechnung der in diesem Kapitel erläuterten Metriken (ausgenommen des Maintainability Indexes) verwendet wird.

2.5.1 Raw Metrics

Bei den Raw Metrics [44] handelt es sich um klassische Software-Metriken, die die Grundlage für die Berechnung weiterer Software-Metriken, wie der Cyclomatic Complexity oder dem Maintainability Index, sind. Durch ihre einfache Verständlichkeit bieten sie eine schnelle und objektive Möglichkeit erste Anhaltspunkte für Optimierungen identifizieren zu können.

$$LOC = SLOC + SLCOM + MULTI + BLANK \quad (1)$$

LOC ... Gesamtanzahl der Codezeilen (= Lines Of Code)

SLOC ... Anzahl der tatsächlichen Codezeilen (= Source Lines Of Code)

SLCOM ... Anzahl der Zeilen, die ausschließlich Kommentare enthalten (= Single Line Comments)

MULTI ... Anzahl der Zeilen, die mehrzeilige Strings enthalten (= Multi Line Strings)

BLANK ... Anzahl der Leerzeilen (= Blank Lines)

$$CPL = \left(\frac{Comments}{LOC} \right) \cdot 100 \quad (2)$$

CPL ... Verhältnis der Kommentarzeilen zur Gesamtanzahl der Zeilen [in %]

(= Comment Per Lines)

$$CPS = \left(\frac{Comments}{SLOC} \right) \cdot 100 \quad (3)$$

CPS ... Verhältnis der Kommentarzeilen zur Anzahl der Quellcodezeilen [in %]

(= Comment Per Source Lines)

$$CMPL = \left(\frac{Comments + Multi}{LOC} \right) \cdot 100 \quad (4)$$

CMPL ... Verhältnis der Zeilen mit Kommentaren und mehrzeiligen Strings zur Gesamtanzahl [in %]

(= Comment & Multi Line Per Lines)

Darüber hinaus kann es hilfreich sein, die folgenden Kennwerte ebenfalls auszuwerten:

- Anzahl der verwendeten Dateien innerhalb eines Systems
- Anzahl der Funktion innerhalb eines Systems
- Anzahl der privaten Methoden innerhalb eines Systems
- Anzahl der öffentlichen Methoden innerhalb eines Systems
- Anzahl der Klassen innerhalb eines Systems
- Anzahl der Imports/Packages
- Durchschnittliche Anzahl der Codezeilen pro Funktion/Methode

2.5.2 Cyclomatic Complexity

Bei der Cyclomatic Complexity (CC) handelt es sich um eine Software-Metrik, die die Komplexität eines Programmes auf Basis der linear unabhängigen Pfade innerhalb eines Programmcodes misst. Die Metrik wurde im Jahre 1976 von Thomas J. McCabe [45] entwickelt und wird deshalb auch oft als McCabe-Metrik bezeichnet. Sie hilft Entwicklern dabei Code objektiv und quantitativ zu bewerten, um sinnvolle Aussagen über Wartbarkeit und Testbarkeit zu treffen. Dabei deutet ein niedriger Wert auf eine geringe und ein hoher Wert auf eine hohe Komplexität hin. Grundsätzlich gibt es zwei Möglichkeiten die zyklomatische Komplexität zu berechnen:

1. Grafisch mithilfe eines Kontrollflussdiagramms:

$$v(G) = e - n + 2p \quad (5)$$

$v(G)$... Zykłomatische Komplexität im Graphen G

e ... Anzahl der Kanten im Graphen G (= Verbindungen zwischen den Knoten)

n ... Anzahl der Knoten im Graphen G

p ... Anzahl der zusammenhängenden Komponenten im Graphen G

Um die Anwendung der grafischen Methode zu demonstrieren wurde folgender Programmcode ausgewählt:

```
def main():
    x = 5
    y = 10

    if x > y:
        print("x is greater.")
    else:
        print("y is greater.")
```

Das dazugehörige Kontrollflussdiagramm sieht folgendermaßen aus:

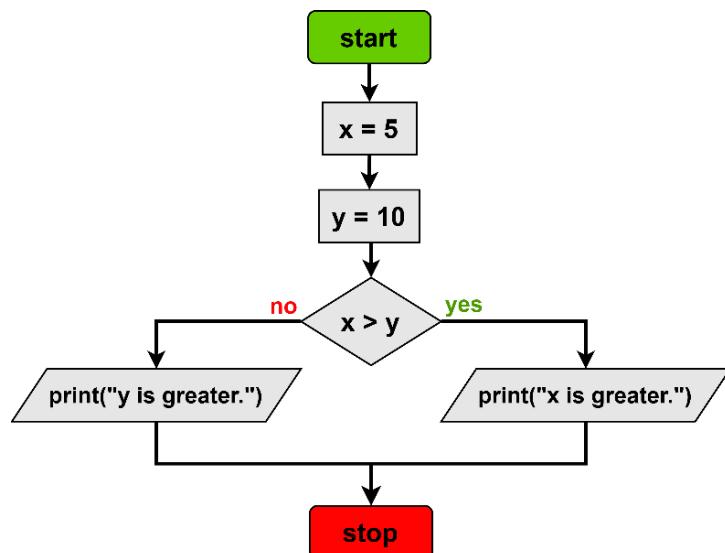


Abbildung 28: Kontrollflussdiagramm G des gegebenen Python-Skriptes

Die Formen des Kontrollflussdiagrammes sind nicht willkürlich gewählt, sie folgen nämlich der ISO/IEC 5807:1985 [46]. Jener Standard besagt folgendes:

- *Start- und Endknoten* → werden mit abgerundeten Rechtecken dargestellt.
- *Prozessschritte* → werden mit Rechtecken dargestellt.
- *Entscheidungspunkte* → werden mit Rauten dargestellt.
- *Eingabe/Ausgabe* → werden mit Parallelogrammen dargestellt.

Die Anzahl der Kanten entspricht der Summe der schwarzen Verbindungslienien:

$$e = 7 \quad (6)$$

Die Anzahl der Knoten entspricht der Summe der grünen, grauen und roten Formen:

$$n = 7 \quad (7)$$

Die Anzahl der zusammenhängenden Komponenten beträgt eins, da es nur eine einzige Kontrollstruktur gibt:

$$p = 1 \quad (8)$$

Das Einsetzen der Gleichungen (6), (7) und (8) in (5) liefert schließlich:

$$v(G) = 7 - 7 + 2 \cdot (1) = 2 \quad (9)$$

2. Rechnerisch mithilfe vereinfachter McCabe-Formel:

Der grafische Ansatz zur Ermittlung der zyklomatischen Komplexität ist zwar hilfreich, um die zugrundeliegende Theorie zu verstehen, ist jedoch sehr aufwendig und kaum praktikabel. Aus genannten Gründen hat sich Mills [47] dazu entschlossen, den Fokus auf strukturierte Programme ohne unstrukturierte Sprünge zu legen. Er zerlegte das Kontrollflussdiagramm in verschiedene Knoten und leitete neue Formeln her:

- θ ... Anzahl der Funktionsknoten (= Anweisungen bzw. Operationen)
- π ... Anzahl der Prädikatsknoten (= Entscheidungspunkte bzw. Bedingungen)
- γ ... Anzahl der Sammelknoten (pro Prädikatsknoten genau ein Sammelknoten)
- Ein- und Ausgangsknoten

Die Anzahl der Kanten ergibt durch die Summe folgender Kanten:

- Die Kante zwischen Eingangsknoten bzw. ersten Funktionsknoten (**1**).
- Die Kanten, die zu den Funktionsknoten führen ($\theta \rightarrow$ je Knoten eine Kante).
- Die Kanten, die von den Prädikatsknoten ausgehen (**3π**). Der Faktor drei ergibt sich aus der eingehenden Kante (vor der Entscheidung), der ausgehenden Kante für den „true“-Pfad und der ausgehenden Kante für den „false“-Pfad.

$$e = 1 + \theta + 3\pi \quad (10)$$

Die Anzahl der Knoten ergibt durch die Summe aller Knoten:

- Funktionsknoten (θ)
- Prädikatsknoten (π)
- Sammelknoten (γ)
- Ein- und Ausgangsknoten (2)

$$n = \theta + \pi + \gamma + 2 \quad (11)$$

Da es aber für jeden Prädikatsknoten genau einen Sammelknoten gibt, ergibt sich:

$$n = \theta + 2\pi + 2 \quad (12)$$

Fundamental für die Vereinfachung der zyklomatische Komplexität nach McCabe (5) ist die Annahme, dass es sich bei der Berechnung um eine einzige Komponente handelt. Das bedeutet:

$$p = 1 \quad (13)$$

Das Einsetzen von Gleichung (10), (11), (12)**Fehler! Verweisquelle konnte nicht gefunden werden.** und (13) in (5) liefert schließlich:

$$v(G) = (1 + \theta + 3\pi) - (\theta + 2\pi + 2) + 2 \cdot (1) = \pi + 1 \quad (14)$$

Folgende Tabelle zeigt den Zusammenhang zwischen verschiedenen Programmkonstrukten in Python und deren Beitrag zur Anzahl der Prädikatsknoten.

Tabelle 7: Einfluss verschiedener Programmkonstrukten auf die zyklomatische Komplexität [44]

Programmkonstrukt π	Begründung
if	+1 An if statement is a single decision.
elif	+1 The elif statement adds another decision.
else	+0 The else statement does not cause a new decision. The decision is at the if.
for	+1 There is a decision at the start of the loop.
while	+1 There is a decision at the while statement.
except	+1 Each except branch adds a new conditional path of execution.
finally	+0 The finally block is unconditionally executed.
with	+1 The with statement roughly corresponds to a try/except block (see PEP 343 for details).
assert	+1 The assert statement internally roughly equals a conditional statement.
Comprehension	+1 A list/set/dict comprehension is equivalent to a for loop.
Boolean Operator	+1 Every boolean operator (and, or) adds a decision point.

Die Demonstration der rechnerischen Methode erfolgt mit demselben Programmcode:

$$\pi = 1 \text{ (da ein if-Statement)} \quad (15)$$

Das Einsetzen von Gleichung (15) in (14) liefert letztlich:

$$v(G) = 1 + 1 = \textcolor{blue}{2} \quad (16)$$

Dies zeigt, dass sowohl der grafische Ansatz (nach McCabe) als auch der rechnerische Ansatz (nach McCabe & Mills) dasselbe Ergebnis ergeben.

Interpretation der Cyclomatic Complexity:

Tabelle 8: Interpretation der zyklomatischen Komplexität [48]

$v(G)$ (CC-Score)	Rang	Risiko	Interpretation
1 - 5	A	Low	Simple block
6 - 10	B	Low	Well structured and stable block
11 - 20	C	Moderate	Slightly complex block
21 - 30	D	More than moderate	More complex block
31 - 40	E	High	Complex block, alarming
41+	F	Very high	Unstable block, error-prone

2.5.3 Halstead Complexity

Bei der Halstead Complexity handelt es sich um eine Software-Metrik, die die Komplexität eines Programmes auf Basis der Operatoren und Operanden innerhalb eines Programmcodes misst. Die Metrik wurde im Jahre 1977 von Maurice H. Halstead [49] entwickelt und trägt daher seinen Namen. Ihr Ziel ist es, eine objektive und quantitative Bewertung von Programmcode in Bezug auf Komplexität, Entwicklungsaufwand und Fehleranfälligkeit zu ermöglichen.

Berechnung der Halstead Complexity:

$$\eta = \eta_1 + \eta_2 \quad (17)$$

η ... Programmavokabular

η_1 ... Anzahl an einzigartigen Operatoren

η_2 ... Anzahl an einzigartigen Operanden

$$N = N_1 + N_2 \quad (18)$$

N ... Programmlänge

N_1 ... Gesamtzahl an Operatoren

N_2 ... Gesamtzahl an Operanden

$$\widehat{N} = \eta_1 \cdot \log_2(\eta_1) + \eta_2 \cdot \log_2(\eta_2) \quad (19)$$

\widehat{N} ... Berechnete Programmlänge

$$V = N \cdot \log_2(\eta) \quad (20)$$

V ... Volumen (= Volume)

$$D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2} \quad (21)$$

D ... Schwierigkeit (= Difficulty)

$$E = D \cdot V \quad (22)$$

E ... Aufwand (= Effort)

$$T = \frac{E}{18} \quad (23)$$

T ... Benötigte Zeit, um den Code zu verstehen (= Time) [s]

$$B = \frac{V}{3000} \quad (24)$$

B ... Anzahl der ausgelieferten Fehler (= Bugs)

Bei der Verwendung der Halstead Complexity zum Vergleich von Programmen, die in verschiedenen Programmiersprachen geschrieben sind, sei Vorsicht geboten. Während Sprachen mit kürzerer Syntax, wie Python, geringere Halstead-Werte aufweisen, können Sprachen mit längerer Syntax, wie C++, höhere Werte verursachen, ohne, dass der Code tatsächlich komplexer ist. Diese Diskrepanzen sind auf die Sprachparadigmen, die eingebauten Funktionen und die Typisierung (dynamisch vs. statisch) der jeweiligen Programmiersprache zurückzuführen. [50]

2.5.4 Maintainability Index

Bei dem Maintainability Index handelt es sich um eine Software-Metrik, die versucht die Wartbarkeit eines Programmes auf Basis der Zeilenanzahl, der Halstead Complexity und der Cyclomatic Complexity zu messen. Die Metrik wurde im Jahre 1992 von P. Oman und J. Hagemeister [51] entwickelt. Das Ziel dieser Metrik ist es Code objektiv und quantitativ zu bewerten, um sinnvolle Aussagen über Wartbarkeit zu treffen. Während ein hoher Maintainability Index bedeutet, dass der Programmcode einfach zu warten ist, weist ein niedriger Index darauf hin, dass der Code schwer verständlich und anfällig für Fehler ist.

Berechnung & Entwicklung des Maintainability Index:

Die ursprüngliche Gleichung lautete wie folgt:

$$MI = 171 - 5.2 \cdot \ln(V) - 0.23 \cdot G - 16.2 \cdot \ln(L) \quad (25)$$

V ... Volumen (nach Halstead)

G ... Zyklomatische Komplexität (nach McGabe & Mills)

L ... Anzahl an Quellcodezeilen (SLOC)

Die Koeffizienten der Gleichung wurden auf Basis von verschiedenen Softwaresystemen von Hewlett-Packard ermittelt bzw. kalibriert. Eine detaillierte Erläuterung der Kalibrierung wurde von Coleman [53][54], Pearse [55] und Welker [56] durchgeführt.

Im Jahre 1997 wurde die Gleichung vom Software Engineering Institute der Carnegie Mellon University erweitert [52]:

$$MI = 171 - 5.2 \cdot \ln(V) - 0.23 \cdot G - 16.2 \cdot \ln(L) + 50 \cdot \sin(\sqrt{2.4} \cdot C) \quad (26)$$

C ... Anteil an Zeilen mit Kommentaren (in rad)

Die ursprüngliche Gleichung berücksichtigt nun die optimierte Wartbarkeit aufgrund der einfach nachvollziehbaren Dokumentation in Form von Kommentaren. Anzumerken ist hier jedoch, dass dies auch tatsächlich nur aussagekräftige bzw. hilfreiche Kommentare einschließt, weshalb eine manuelle Verifikation erforderlich ist.

Im Jahre 2007 wurde die Gleichung von Microsoft [57] wieder angepasst:

$$MI = \max \left[0, 100 \cdot \frac{171 - 5.2 \cdot \ln(V) - 0.23 \cdot G - 16.2 \cdot \ln(L)}{171} \right] \quad (27)$$

Da das Ergebnis der Formel vorher von einem undefinierten negativen Wert hin zu 171 reichte wurden folgende Anpassungen getroffen:

- Der Divisor 171 wurde eingeführt, um den Wertebereich auf eine verständlichere Skala von 0 bis 100 zu normieren.
- Der Minimalwert wurde auf 0 begrenzt, da Werte nahe 0 bereits als schwer wartbar galten.

Die derzeit aktuelle Gleichung wurde im Jahre 2012 von Radon [40] aufgestellt und lautet folgendermaßen:

$$MI = \max \left[0, \min \left[100, 100 \cdot \frac{171 - 5.2 \cdot \ln(V) - 0.23 \cdot G - 16.2 \cdot \ln(L) + 50 \cdot \sin(\sqrt{2.4} \cdot C)}{171} \right] \right] \quad (28)$$

Sie nutzt die Gleichung von Microsoft [57] als Grundlage und ergänzt diese um den Kommentar-Term vom Software Engineering Institute der Carnegie Mellon University [52].

Im Rahmen dieser Arbeit wird der Kommentar-Term durch einen Docstring-Term ersetzt.

$$D = \frac{\text{Anzahl der vorhandenen Docstrings}}{\text{Anzahl an Modulen} + \text{Anzahl an Klassen} + \text{Anzahl an Methoden/Funktionen}} \quad (29)$$

$$MI = \max \left[0, \min \left[100, 100 \cdot \frac{171 - 5.2 \cdot \ln(V) - 0.23 \cdot G - 16.2 \cdot \ln(L) + 50 \cdot \sin(\sqrt{2.4} \cdot D)}{171} \right] \right] \quad (30)$$

D ... Anteil an verwendeten Docstrings (in rad)

Interpretation des Maintainability Index:

Tabelle 9: Interpretation des Maintainability Index [48]

MI-Score	Rang	Wartbarkeit
20 - 100	A	Very high
10 - 19	B	Medium
0 - 9	C	Extremely Low

Kritik am Maintainability Index:

Doch bevor der Maintainability Index verwendet wird, sollen möglich Probleme erläutert werden. Im Jahre 2014 hinterfragte Van Deursen [58] die Nützlichkeit des Maintainability Index und äußerte folgende Kritik:

- Die zugrundliegenden Parameter (V , G , L und C) korrelieren mit der Zeilenanzahl, weshalb einfachere Messungen, wie die durchschnittliche Zeilenanzahl pro Modul, zielführender sein könnten.
- Die Formel wurde auf Basis einer kleinen Stichprobe von C- bzw. Pascal-Programmen entwickelt und könnte Charakteristiken widerspiegeln, die von modernen objektorientierten Programmiersprachen, wie Python oder JavaScript, abweichen.

Ähnliche Feststellungen machten auch Sjøberg, Anda und Mockus im Rahmen ihrer empirischen Studie [59], in welcher sie eine gleichwertige Anwendung von vier unabhängigen Unternehmen analysierten. Obwohl die Entwicklungskosten mit €18.000, €25.000, €52.000 und €61.000 stark variierten, stellten sie fest, dass höhere Budgets trotz erhöhtem Entwicklungsaufwand keinen klaren Einfluss auf die Wartbarkeit hatten. Lediglich die Größe des Systems, also die Anzahl an verwendeten Modulen, und die inverse Kohäsion wiesen eine eindeutige Korrelation mit dem tatsächlichen Wartungsaufwand auf. Eine hohe inverse Kohäsion impliziert eine geringe Kohäsion, was bedeutet, dass die Komponenten eines Moduls funktional kaum zusammenhängen und dadurch die Wartung erschweren. Damit bestätigten sie die Mutmaßungen von M. Riaz, M. Mendes und E. D. Tempero [60].

Deswegen ist es wichtig zu beachten, dass der Maintainability Index eine **experimentelle Metrik** darstellt und stets in Kombination mit anderen Metriken interpretiert werden sollte. Darüber hinaus ist bei der Verwendung der Formal nach Radon [44] eine **manuelle Verifikation der Sinnhaftigkeit der Kommentare bzw. Docstrings essenziell!**

2.6 Allgemeine Software-Prinzipien

Bevor auf die spezifischen Clean-Code-Prinzipien in Python eingegangen wird, sollen zur besseren Nachvollziehbarkeit auch allgemeine Software-Prinzipien erläutert werden. Die vermutlich prägnanteste und fundamentalste Quelle ist "Seven Principles of Software Development" [61], welche von David Hooker im Jahre 1996 verfasst wurde. Seine Erkenntnisse wurden beispielweise auch von Pressman in der neunten Ausgabe von "Software Engineering: A Practitioner's Approach" [12] im Jahre 2019 zitiert.

Hooker definiert sieben grundlegende Prinzipien, die die Softwareentwicklung als Ganzes betrachten und unabhängig von der verwendeten Programmiersprache gelten. Diese lauten wie folgt:

2.6.1 The Reason It All Exists [62]

Problem: Während des Softwareentwicklungsprozesses ist es schwierig, Entscheidungen zu treffen.

Kontext: Tagtäglich stehen Entwickler vor ungewissen Fragen und müssen sich deshalb mit der Entscheidungsfindung im Rahmen der Softwareentwicklung beschäftigen. Diese Fragen beinhalten mitunter:

- "Wie lauten die Anforderungen?"
- "Welche Methodik soll verwendet werden?"
- "Welche Programmiersprache ist am besten?"
- "Wer ist die Zielgruppe bzw. wer ist der Endbenutzer?"

Damit ein Projekterfolg gewährleistet werden kann, sollten stets die bestmöglichen Entscheidungen getroffen werden.

Lösung: Ein Softwaresystem existiert genau aus einem Grund, und zwar, um den Endbenutzern einen echten Mehrwert zu bieten. Hier bedarf es deshalb einer Definition des Synonyms "echter Mehrwert" (engl. "Real Value") [63]: Ein echter Mehrwert umfasst alle Faktoren, welche objektiv zur Zielerreichung beitragen und anhand von greifbaren Ergebnissen bestätigt werden können. Eine Kosteneinsparung, eine Leistungssteigerung oder eine andere messbare Verbesserung sind Beispiele solcher konkreten Ergebnisse. Sämtliche Entscheidungen sollten immer auf Basis dieses Grundsatzes getroffen werden. Bevor eine neue Anforderung spezifiziert, eine Systemkomponente konzipiert oder eine Funktionalität festgelegt wird, sollte folgende Frage gestellt werden: "Bringt dies wirklich einen echten Mehrwert für das System?". Lautet die Antwort darauf "Nein", dann sollte eine Implementierung oder Umsetzung stets vermieden werden.

Resultat: Alle Entscheidungen werden korrekt sein, und nur jene Aspekte, die einen echten Mehrwert bieten, werden im entwickelten System vorhanden sein.

2.6.2 KISS (Keep It Short & Simple) [64]

Problem: Die Nachvollziehbarkeit eines Systems ist durch die kontinuierlich zunehmende Komplexität erschwert.

Kontext: Eine erhöhte Komplexität zieht mehr Fehler mit sich und verursacht dadurch einen Mehraufwand im Bereich der Wartung. Optimalerweise werden Systeme so erstellt, dass sie eine verbesserte Verständlichkeit, Wartbarkeit, Flexibilität und eine geringe Fehleranfälligkeit aufweisen.

Lösung: Die Softwareentwicklung ist kein willkürlicher Prozess, denn es gibt einige Faktoren, die bei der Konzeptionierung zu berücksichtigen sind. Bereits das Konzept sollte so einfach wie möglich gestaltet werden, um daraus ein verständliches und wartbares System erstellen zu können. Das bedeutet aber nicht, dass im Sinne der Einfachheit auf Funktionalitäten oder interne Strukturen verzichtet werden soll. Einfachheit bedeutet außerdem nicht schnell und mangelhaft (engl. "quick and dirty") zu handeln, sondern ganz im Gegenteil, sie erfordert oft ein grundlegendes Verständnis und mehrere sorgfältige Iterationen.

Resultat: Eine Software, die wartbar, verständlich und weniger fehleranfällig ist.

2.6.3 Maintain the Vision [61]

Problem: Ohne einer klaren Vision droht ein Softwareprojekt sich zu einem Patchwork aus widersprüchlichen Ideen und Entwürfen zu entwickeln.

Kontext: In komplexen und großen Softwareprojekten ist häufig eine Vielzahl von Entwicklern involviert, die andere Interessen aufweisen und unterschiedliche Vorstellungen vom Endprodukt haben. Damit das Projektteam dasselbe Ziel verfolgt und keine Abweichungen von der ursprünglichen Idee geschehen, ist eine zentrale Vision essenziell.

Lösung: Für ein erfolgreiches Softwareprojekt ist es wichtig vorab eine klare Vision zu definieren und diese beständig beizubehalten. Im Idealfall sollte dies durch einen erfahrenen Software-Architekten erfolgen. Jene Kompromisse, die die konzeptionelle Integrität des Projektes gefährden, sollten, wenn möglich, vermieden werden.

Resultat: Ein konsistentes und robustes Softwaresystem, welches problemlos wartbar, erweiterbar und benutzbar ist.

2.6.4 What You Produce, Others Will Consume [65]

Problem: Eine Vielzahl an entwickelter Software ist kaum bis gar nicht benutzbar, verstehtbar, wartbar und/oder erweiterbar.

Kontext: Es ist kaum der Fall, dass eine zuverlässige industrierelevante Software im Alleingang von einer einzigen Person entwickelt und verwendet wird. Viel wahrscheinlicher ist es, dass sich früher oder später jemand anderes am Projekt beteiligen wird. Unabhängig davon, ob die beizutretende Person eine Dokumentation erstellen bzw. erweitern oder die Software warten, verbessern bzw. nutzen möchte, muss der Erstentwickler ein fundamentales Grundverständnis der Software ermöglichen.

Lösung: Die Spezifizierung, die Konzeptionierung und die Implementierung sollten stets mit dem Hintergedanken erfolgen, dass jemand anderes darauf angewiesen ist, die Software nachvollziehen zu müssen. Um die Anforderungen an die Software möglichst optimal zu definieren, ist die Berücksichtigung der jeweiligen Zielgruppen essenziell. Die Spezifizierung liegt im Interesse der Endbenutzer, die Konzeptionierung im Interesse der Entwickler und die Implementierung im Interesse derjenigen, die die Software warten, verbessern und erweitern müssen.

Resultat: Eine Software, die maßgeblich zu einer vereinfachten Nutzung und Wartung beiträgt. Neben den Endbenutzern, welche die finalen ausführbaren Applikationen verwenden, wird es auch der Fall sein, dass andere Entwickler den zugrundeliegenden Code verändern und daher auch debuggen müssen. Ersparen sich diese Mitarbeiter deshalb Aufwand bzw. Zeit, so steigert dies letztlich auch den Wert des Gesamtsystems.

2.6.5 YAGNI (You Aren't Gonna Need It) [66]

Problem: Wie können aktuelle Anforderungen sinnvoll umgesetzt werden, sodass potenzielle zukünftige Anforderungen nicht vernachlässigt werden?

Kontext: Bei der Erstellung oder Erweiterung eines Softwaresystems möchten Entwickler nicht nur die derzeitigen Anforderungen einhalten, sondern auch zukünftige Flexibilität einplanen. Spätere Anforderungen sind jedoch meist unbekannt bzw. unklar und ändern sich im Laufe eines Projektes oftmals. Eine Bemühung diese ungewissen Anforderungen zu berücksichtigen, führt oft zu überkomplizierten oder gar unvollständigen Systemkomponenten.

Lösung: Es sollte immer nur der tatsächlich benötigte Quellcode auf Basis der derzeitigen Anforderungen geschrieben werden. Das YAGNI-Prinzip ist eine Überarbeitung bzw. Korrektur des ursprünglichen „Be Open to the Future“-Ansatzes [67].

Resultat: Ein Softwaresystem, welches die aktuellen Bedürfnisse erfüllt und bei Bedarf auf zukünftige Anforderungen angepasst werden kann. Eine spätere Erweiterung ist meist kosteneffizienter als vorab einen erheblichen Aufwand zu betreiben.

2.6.6 Plan Ahead for Reuse [61]

Problem: Ohne ausreichender Planung und transparenter Kommunikation kann ein Softwaresystem die Vorteile der Wiederverwendbarkeit, wie Zeit- oder Arbeitsersparnis, nicht optimal ausnutzen.

Kontext: Obwohl die Wiederverwendbarkeit einige Vorteile mit sich zieht, ist eine dementsprechende Implementierung sehr anspruchsvoll und erfordert eine sorgfältige Planung.

Lösung: Systemkomponenten und Funktionen sollten gezielt wiederverwendbar gestaltet, dokumentiert und intern kommuniziert werden, sodass das Projektteam den Nutzen kennt. Darüber hinaus kann der Quellcode durch Einführung von Entwurfsmustern in der Entwicklungsphase optimiert werden. Zudem bietet sich auch der Einsatz einer objektorientierten Programmiersprache an.

Resultat: Eine vorab geplante Wiederverwendung erspart Kosten und steigert den Wert der entwickelten Systemkomponenten und des Systems, in welches sie eingebaut werden. Dies gestaltet künftige Softwareentwicklungsprozesse effizienter und nachhaltiger.

2.6.7 Think [61]

Problem: Wenn Entscheidungen getroffen werden, ohne davor klare Überlegungen zu treffen, führt dies häufig zu Fehlern bzw. ineffizienten Lösungen.

Kontext: In der schnelllebigen Softwareentwicklung steht der Projektfortschritt oft im Vordergrund und fundamentale Überlegungen werden vernachlässigt. Durch bewusste Reflexion vor der tatsächlichen Umsetzung können qualitativ wesentlich hochwertigere Lösungen entstehen.

Lösung: Ein gezieltes Nachdenken, in Form von einer grundlegenden Identifikation und Recherche von Wissenslücken, erlaubt es, bessere Ergebnisse zu erzielen. Sollte die Lösung dennoch Probleme aufweisen, können wertvolle Erfahrungswerte gesammelt werden, welche idealerweise zu einer kontinuierlichen Verbesserung beitragen.

Resultat: Entweder ein durchdachtes und hochwertiges Softwaresystem oder der Gewinn von nützlichen Erkenntnissen, welche für die fortlaufende Weiterentwicklung verwendet werden können.

2.7 Large Language Model

2.7.1 Definition

Ein Large Language Model (LLM), zu Deutsch großes Sprachmodell, ist ein Deep Neural Network (DNN), welches menschenähnliche Texte verstehen und generieren kann. Der Begriff „Large“ bezieht sich dabei sowohl auf die enorme Größe des Modells und der Vielzahl an Parametern als auch auf die immense Menge an Daten, mit welchen es trainiert wird. Solche Modelle besitzen typischerweise mehrere zehn bis hundert Milliarden Parameter (anpassbare Gewichtungen innerhalb des neuronalen Netzwerks), welche präzise Vorhersagen über das nächste Wort in einer Sequenz ermöglichen. Wird dieser Prozess mehrmals durchlaufen, entstehen aus einzelnen Wörtern ganze Sätze, daraus wiederum zusammenhängende Absätze und in weiterer Folge sogar kohärente Texte. Da LLMs somit in der Lage sind neue Texte erstellen zu können, stellen sie eine Form der generativen künstlichen Intelligenz (Generative AI oder GenAI) dar. [68]

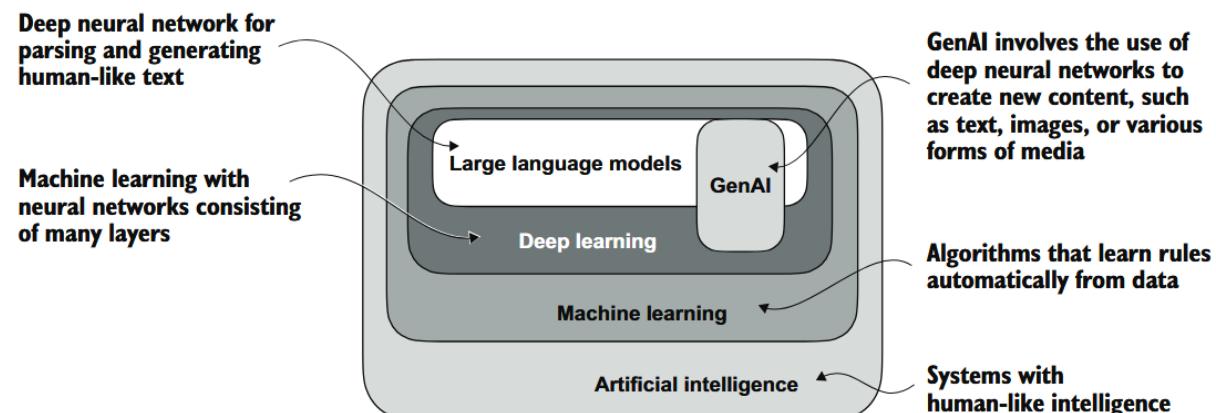


Abbildung 29: Hierarchische Struktur der Künstlichen Intelligenz [68]

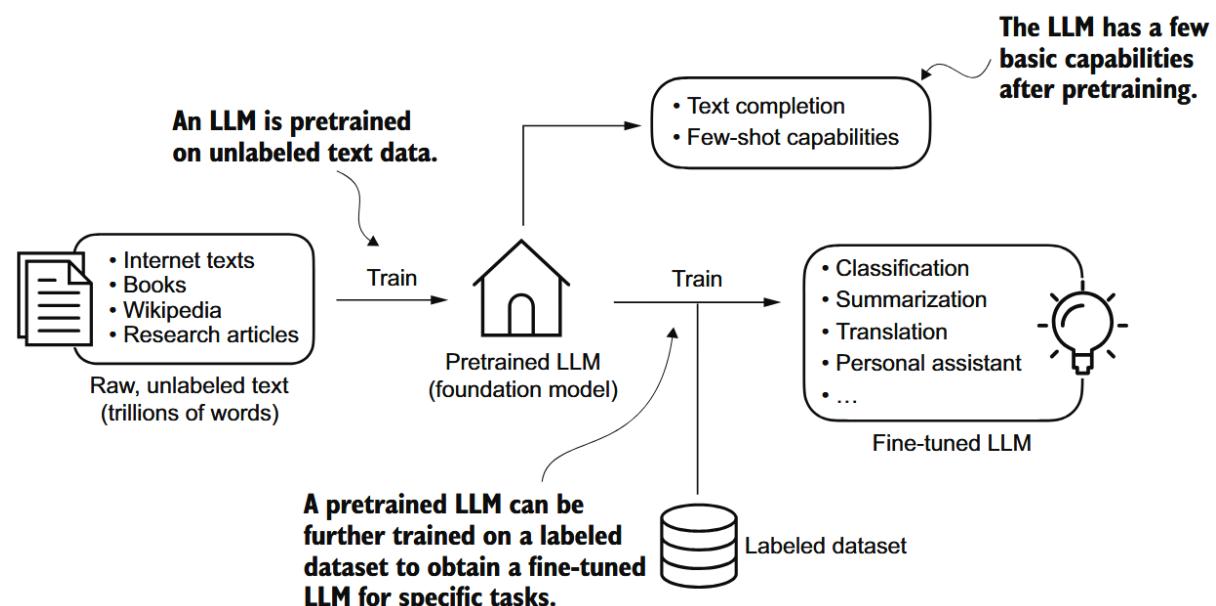


Abbildung 30: Vom Pretraining bis zum Fine-Tuning eines LLMs [68]

Das wohl bekannteste LLM ist ChatGPT, welches am 30. November 2022 [6] veröffentlicht wurde. Der Name setzt sich dabei aus den folgenden Bestandteilen zusammen:

- **Chatbot** → Interaktive Kommunikation im Chat-Format.
- **Generative** → Eigenständige Erstellung neuer Texte.
- **Pre-trained** → Vorab auf umfangreichen Daten trainiert.
- **Transformer** → Neuronale Netzwerkarchitektur mit Self-Attention-Mechanismus zur effizienten Verarbeitung von Sprache und semantischen Beziehungen.

2.7.2 Funktionsweise

Fundamental für heutige LLMs ist die Transformer-Architektur, die im Jahre 2017 von Google entwickelt [68] wurde. Diese Architektur führte durch die Einführung eines Self-Attention-Mechanismus zu einem Paradigmenwechsel in der Verarbeitung natürlicher Sprache (NLP). Dieser Mechanismus ermöglichte es globale Abhängigkeiten zwischen Tokens innerhalb eines Textes effizienter zu erfassen und parallel zu verarbeiten. Da dies die Rechenleistung erheblich optimierte, wurden zuvor eingesetzte sequenzielle Recurrent Neural Networks (RNNs) und Long Short-Term Memory Networks (LSTMs) weitgehend ersetzt. Um den Rahmen dieser Arbeit nicht zu sprengen, werden weitere Aspekte der Transformer-Architektur nicht im Detail erläutert. Eine graphische Darstellung samt Erklärung ist [68] zu entnehmen.

Folgende Abbildung veranschaulicht, wie ein LLM das nächste Wort einer gegebenen Sequenz vorhersagt. Dabei wird der Eingabetext in Tokens aufgeteilt, in Embeddings konvertiert, durch ein neuronales Netz verarbeitet und schließlich durch eine Softmax-Funktion in eine Wahrscheinlichkeitsverteilung umgewandelt.

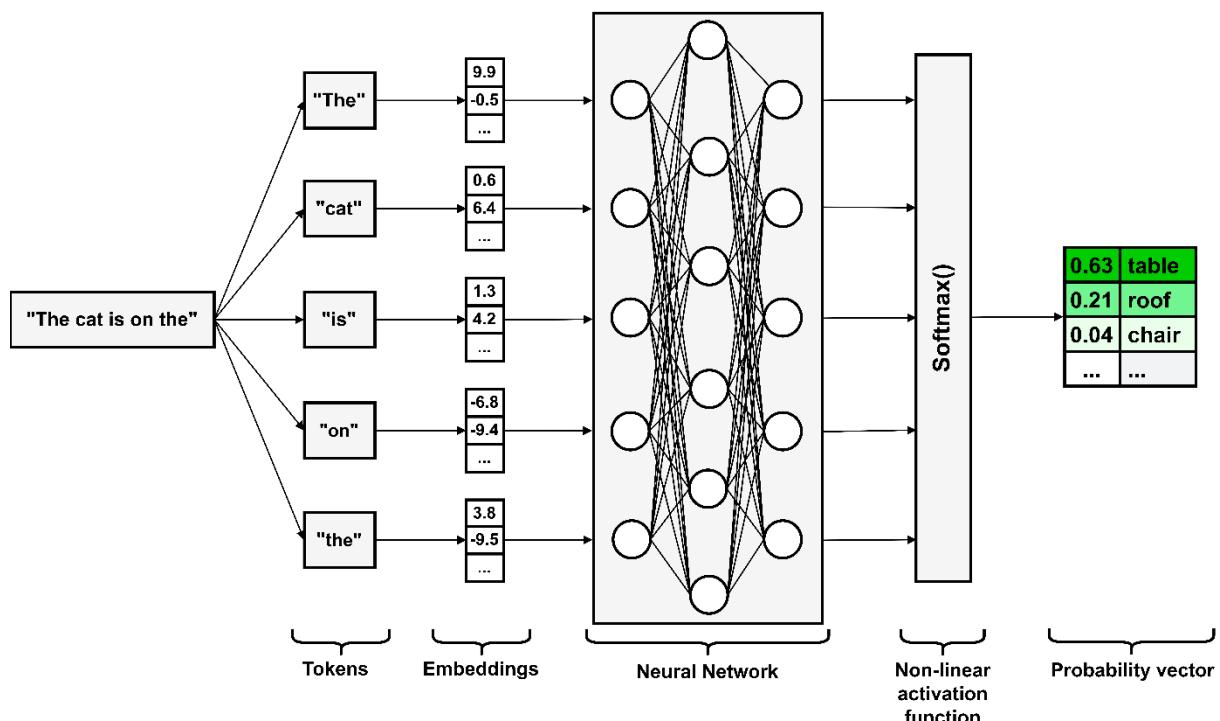


Abbildung 31: Vorhersage des wahrscheinlichsten nächsten Wortes in einem LLM [69]
(eigene Abbildung)

Tokens

Bei der Tokenization wird der eingegebene Roh-Text, in diesem Fall „The cat is on the“, in mehrere kleine Teile, sogenannte Tokens, aufgeteilt. Das Ziel ist es, den Text in ein Format zu bringen, welches effizient von Machine-Learning Modellen verarbeitet werden kann. Wie der String konkret aufgeteilt wird, hängt dabei vom verwendeten Tokenizer ab. [68]

Tabelle 10: Übersicht über Tokenizer-Typen, ihre Anwendungen und Beispiele

Tokenizer	Anwendung	Beispiel
Character-Level	OCR (Tesseract)	Text: "Hello" Tokens: ["H", "e", "l", "l", "o"]
Whitespace-Level	SpaCy	Text: "The cat is on the table." Tokens: ["The", "cat", "is", "on", "the", "table."]
Sentence-Level	Google Translate	Text: "Hello! How are you?" Tokens: ["Hello!", "How are you?"]
WordPiece	BERT [70]	Text: "playing" Tokens: ["play", "##ing"]
Byte-Pair Encoding (BPE)	GPT-1 [71]	Text: "Österreich" Initial Tokens: (Unicode character-by-character) ["Ö", "s", "t", "e", "r", "r", "e", "i", "c", "h"] After BPE-Merge: ["Öst", "er", "reich"]
Byte-Level BPE (BBPE)	GPT-2 [72] GPT-3 (r50k_base [73]) GPT-3.5 (cl100k_base [73]) GPT-4 (cl100k_base [73]) GPT-4o (o200k_base [73])	Text: "Österreich" Initial Tokens: (UTF-8 byte-by-byte in decimal) ["C3", "96", "73", "74", "65", "72", "72", "65", "69", "63", "68"] After BPE-Merge: ["Ö", "sterreich"]

Der Grund, warum heutzutage vorrangig BBPE eingesetzt wird, ist, dass es mit 1/8 des Vokabulars eine ähnliche Performance wie BPE liefert. [74]

Für eine detaillierte Erklärung der jeweiligen Tokenizer sind [75][74] und [76] empfehlenswert. Eine benutzerfreundliche Weboberfläche für die Tokenization von Texten in GPT-3-, GPT-3.5- und GPT-4-Modellen ist unter [77] aufrufbar.

Embeddings

Damit diskrete Daten (Wörter, Bilder, etc.) in neuronalen Netzen und in weiterer Folge in LLMs verwendet werden können, müssen sie in numerische Vektoren, sogenannte Embeddings, umgewandelt werden. Bei Token-Embeddings handelt es sich somit um die Darstellung eines

Tokens innerhalb eines Vektorraums. Die Dimension des Vektorraums hängt von der Modellgröße bzw. Architektur ab und reicht von 50 (einfache Word-Embeddings) bis hin zu zehntausend (große Sprachmodelle wie GPT-3). Da ähnliche Tokens ähnliche Vektorrepräsentationen aufweisen, können auf Basis dieser Charakteristik semantische Beziehungen zwischen den verschiedenen Token erkannt werden. [69]

Damit semantische Beziehungen zwischen Embeddings besser aufgefasst werden können, ist ein kurzes Beispiel notwendig:

$$E(\text{Bratwurst}) - E(\text{Germany}) \approx E(\text{Sushi}) - E(\text{Japan}) \quad (31)$$

E ... Token Embedding

Wird Gleichung (31) nach $E(\text{Bratwurst})$ umgestellt, so ergibt sich:

$$E(\text{Bratwurst}) \approx E(\text{Sushi}) - E(\text{Japan}) + E(\text{Germany}) \quad (32)$$

Graphisch in einem exemplarischen Raum dargestellt sieht Gleichung (32) wie folgt aus:

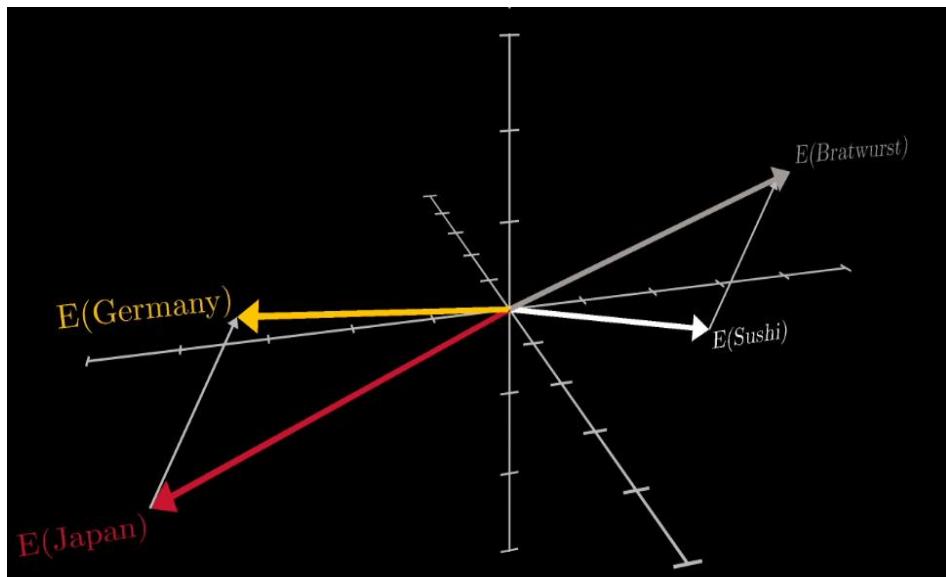


Abbildung 32: Darstellung verschiedener Token-Embeddings als Vektoren [78]

Die Abbildung zeigt, dass das Modell, welches das Token-Embedding erstellt hat, die Beziehung zwischen *Japan* und *Sushi* erkannt hat und durch eine Vektoroperation eine ähnliche Beziehung zwischen *Deutschland* und *Bratwurst* darstellen kann. Weitere Beispiele für die Ermittlung semantischer Beziehungen sind [79] zu entnehmen.

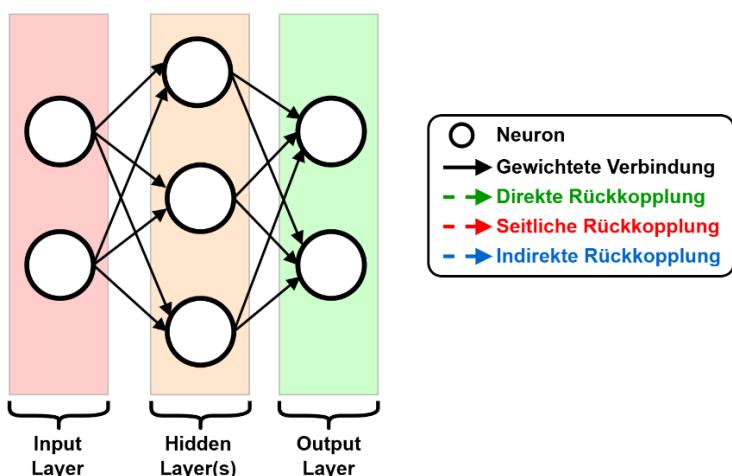
Damit die Position des Tokens in einer gegebenen Sequenz berücksichtigt werden kann, gibt es absolute und relative Positional-Embeddings. OpenAI's GPT-Modelle verwenden beispielweise absolute Positional-Embeddings, welche zu dem jeweiligen Token-Embedding addiert werden. [69]

Neural Network

Ein künstliches neuronales Netz ist eine grundlegende Technologie im Bereich des maschinellen Lernens (Machine Learning – ML). Sie bestehen aus mehreren Schichten, in denen sich künstliche Neuronen befinden und werden verwendet, um komplexe Muster in Daten zu erkennen. Das Netz wird dabei durch die Tiefe (=Anzahl der Schichten) und die Breite (=Anzahl der Neuronen in den einzelnen Schichten) charakterisiert. [80]

Moderne LLMs basieren auf Transformer-Architekturen, die durch ihren Self-Attention-Mechanismus den Einsatz von Feed Forward Neural Networks (FFNN) erlauben. FFNNs verarbeiten Informationen in eine Richtung und differenzieren sich daher deutlich von rückkoppelnden Recurrent Neural Networks (RNN). Ein FFNN minimiert sequentielle Abhängigkeiten, ermöglicht eine Parallelisierung der Berechnungen und damit eine Reduktion der Rechenzeit. [80]

Feed Forward Neural Network



Recurrent Neural Network

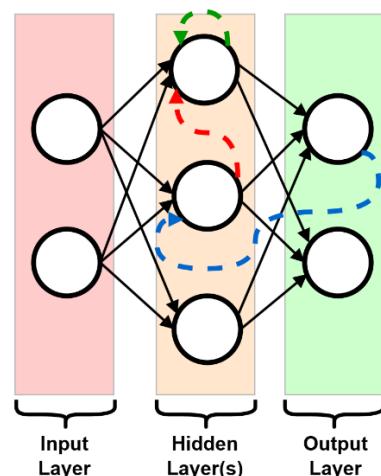


Abbildung 33: Vergleich zwischen Feed Forward Neural Networks und Recurrent Neural Networks
[80] (eigene Abbildung)

Eine Vielzahl an weiteren neuronalen Netzen ist [81] zu entnehmen.

Non linear activation function

Ein Logit [81] ist eine beliebige reelle Zahl, den ein Neuron im Output-Layer eines neuronalen Netzes ausgibt. Um aus den nicht-normalisierten Logits eine Wahrscheinlichkeitsverteilung zu extrahieren, wird die Softmax-Funktion angewandt. Sie stellt sicher, dass jedes Element des Ausgabevektors bzw. Wahrscheinlichkeitsvektors einen Wert zwischen 0 und 1 hat, wobei die Summe aller Elemente stets 1 beträgt.

$$\text{Softmax}(z_i, T) = \frac{e^{\left(\frac{z_i}{T}\right)}}{\sum_{j=1}^K e^{\left(\frac{z_j}{T}\right)}} \quad (33)$$

z_i ... i-tes Element des Logit-Vektors

K ... Anzahl an Elementen im Logit-Vektor

T ... Temperatur (= Hyperparameter)

Mithilfe der Temperatur lässt sich die Wahrscheinlichkeitsverteilung beeinflussen, wodurch die Kreativität des generierten Ausgabetextes angepasst werden kann. Während eine niedrige Temperatur zu deterministischeren Ergebnissen führt, ermöglicht eine hohe Temperatur kreative und vielfältige Ergebnisse. Hier gilt es jedoch die Dokumentation der jeweiligen Anbieter zu kontrollieren, da nicht alle Modelle dieselben Temperaturbereiche verwenden. Typischerweise verwenden Anbieter (OpenAI, Google, Xai) einen Wertebereich von 0 bis 2, jedoch gibt es auch andere Anbieter (Anthropic) die einen Bereich von 0 bis 1 nutzen. Grundsätzlich ist der Wertebereich zwischen 0 und 1 in allen Modellen konsistent. Die Erweiterung der Skala über 1 ermöglicht lediglich eine verstärkte Variabilität, die jedoch selten Anwendung findet.

Tabelle 11: Einfluss der Temperatur auf die Generierung von Texten [83]

Temperatur	Einsatzbereich	Beschreibung
0.0	Übersetzung, Code-Generierung	Konsistent, präzise, vorhersehbar
0.5	Zusammenfassungen, Chatbots	Ausgewogen, verständlich, leicht variabel
1.0	Brainstorming, Storytelling	Abwechslungsreich, kreativ
> 1.0	Nicht empfohlen (Halluzination)	Unvorhersehbar, unstrukturiert, zufällig

Abgesehen von dedizierten Benutzeroberflächen wie OpenAI Playground oder Anthropic Workbench und APIs, verwenden frei zugängliche Webapplikationen wie ChatGPT standardmäßig mittlere Temperatur-Werte (meist 1). Insbesondere wenn Reproduzierbarkeit erwünscht ist, sollte die Temperatur auf null gesetzt werden.

Bei Verwendung einer Temperatur gleich Null würde die Softmax-Funktion eine Division durch Null erzeugen. Da dies mathematisch nicht definiert ist, kann sie in diesem Sonderfall entweder durch eine Zahl nahe Null (z.B.: 0.00001) angenähert oder optimalerweise deterministischer mittels Greedy Decoding durch eine argmax()-Funktion ersetzt werden. Angemerkt sei hier jedoch, dass eine Temperatur von Null aufgrund der begrenzten Zahlengenauigkeit und der damit einhergehenden Rundungsfehler keinen vollständigen Determinismus gewährleistet.

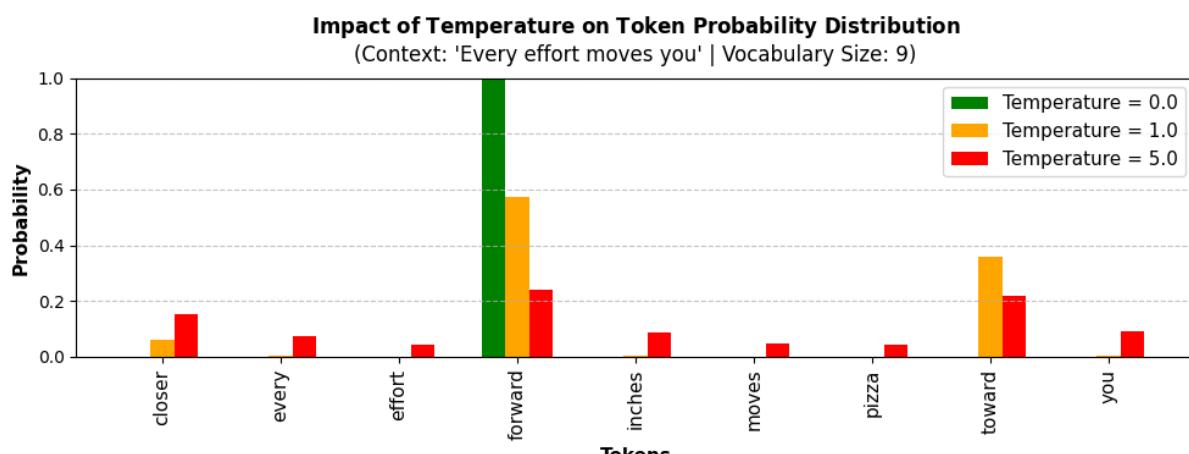


Abbildung 34: Einfluss der Temperatur auf die Wahrscheinlichkeitsverteilung des nächsten Tokens [68] (eigene Darstellung)

Wie sich die Temperatur in einem konkreten Prompt auszeichnet, kann anhand von folgendem Beispiel dargestellt werden:

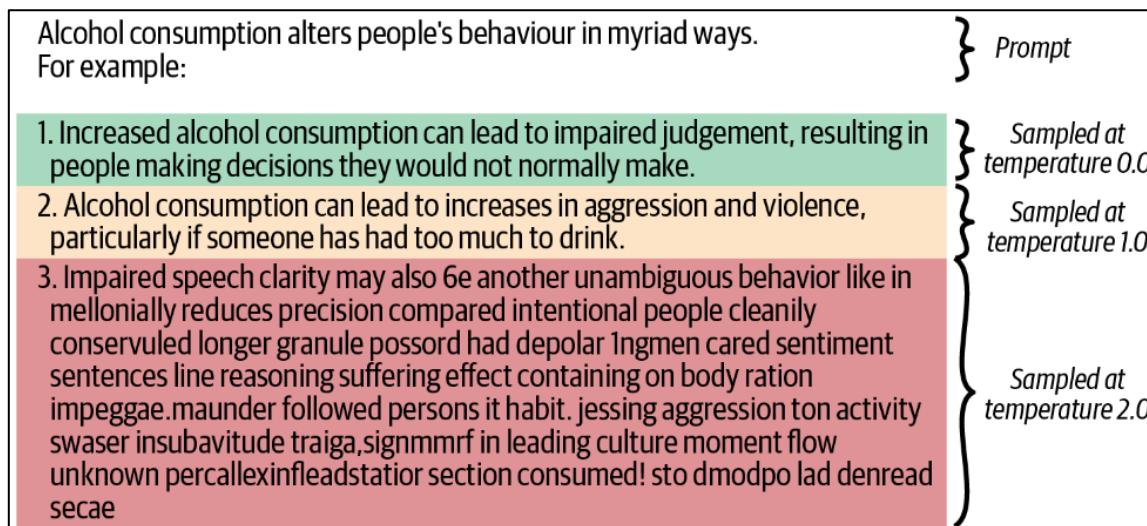


Abbildung 35: Einfluss der Temperatur auf die Ausgabe eines LLMs (OpenAI's text-davinci-003) [83]

2.7.3 Auswahl eines geeigneten Modells

Die Wahl des LLMs basiert auf folgenden Eigenschaften (in absteigender Relevanz) [83]:

- Intelligenz
- Geschwindigkeit
- Kosten
- Benutzerfreundlichkeit
- Funktionalität
- Spezielle Anforderungen (Nicht-kommerziell, Open-Source, Datenresidenz, etc.)

Die Entwicklung leistungsfähigerer LLMs ist im vollen Gange, weshalb das derzeit für den jeweiligen am besten geeignete Modell stark variiert. Daher ist es empfehlenswert, kontinuierlich nach Veröffentlichungen von neuen Modellen Ausschau zu halten.

Folgende Quellen sind für codespezifische Anwendungsfälle geeignet:

- *Aider LLM Leaderboards* [85][68]
- *CanAiCode Leaderboard* [86]
- *ProLLM Benchmarks* [87]

Für eine detailliertere Analyse weiterer Parameter, einschließlich der Intelligenz auf Basis von verschiedenen Indizes, der Ausgabe-Geschwindigkeit, der Ein- und Ausgabe-Preise und der Latenz, ist das Dashboard von *Artificial Analysis* [88] besonders aufschlussreich.

2.7.4 Fine-tuning

Nachdem das effizienteste Modell - typischerweise das kleinstmögliche, aber dennoch ausreichende Modell - ausgewählt wurde, gibt es die optionale Möglichkeit dieses durch Fine-Tuning weiter zu optimieren. Dabei wird das bereits existierende Modell auf die spezifischen Anforderungen der gewünschten Anwendungen trainiert. Grundvoraussetzung dafür ist ein hochwertiger Datensatz, der faktisch korrekt ist, nur relevante Informationen enthält und dem gewünschten Ausgabeformat entspricht. Welche Art von Fine-Tuning empfehlenswert ist, hängt von folgenden Kriterien ab:

Tabelle 12: Vergleich von verschiedenen Fine-Tuning-Methoden [83]

	Lernumfang	Empfohlene Anzahl an Trainingsdokumenten	Trainingsdauer
Full fine-tuning	Komplett neues Wissen in einer völlig neuen Domäne	$> 10^4$	Wochen bis Monate
Parameter efficient fine-tuning	Neue/spezifische Feinheiten in einer bereits bekannten Domäne	$> 10^3$	Tag
Soft prompting	Nur die Informationen aus dem aktuellen Prompt	$> 10^2$	Stunden

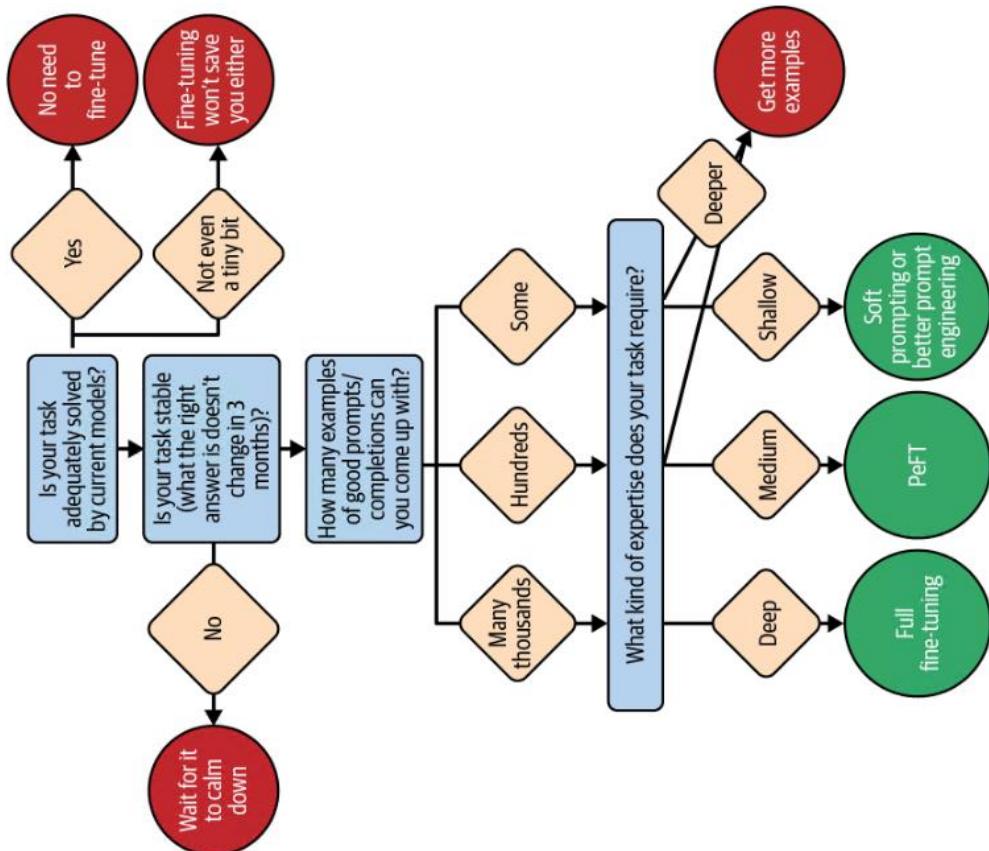


Abbildung 36: Entscheidungsdiagramm zur Notwendigkeit von Fine-Tuning [83]

Die explizite Funktionsweise der verschiedenen Fine-Tuning-Methoden wird im Rahmen dieser Arbeit nicht erläutert. Weiterführende Informationen hierzu befinden sich in [83].

2.8 Prompt Engineering

Prompt Engineering beschreibt den Prozess der Erstellung und Optimierung eines Prompts (=Eingabetextes) für LLMs. Der Prompt spielt dabei eine fundamentale Rolle, da er das Verhalten und die Ausgabe des LLMs steuert. Das Ziel ist es, durch eine sorgfältige Verwendung von spezifischen Wörtern, Symbolen, Phrasen, Beispielen und Strukturen das gewünschte Verhalten hervorzurufen. [69]

In weiterer Folge kann mithilfe eines gut gewählten Prompts und einer Adjustierung weiterer Model-Parameter, beispielweise der Temperatur, eine hilfreiche LLM-basierte Anwendung entwickelt werden. Diese Applikationen bieten eine Vielzahl an realen Anwendungsfällen, die in den verschiedensten Bereichen eingesetzt werden. Dazu zählt unter anderem die Sprachübersetzung von Texten, die Erstellung von Programmcode, die Zusammenfassung von Transkripten, die Unterstützung kreativer Denkprozesse durch Brainstorming und die Erzeugung kreativer Inhalte im Bereich des Storytellings. [68]

Ob sich der Einsatz einer LLM-basierten Anwendung empfiehlt, hängt von der jeweiligen Aufgabe ab. Der Einsatzbereich lässt sich grundsätzlich in zwei Kategorien einteilen: Einerseits in einfache, vielseitig nutzbare Abläufe und andererseits in spezialisierte, komplexe Prozesse:

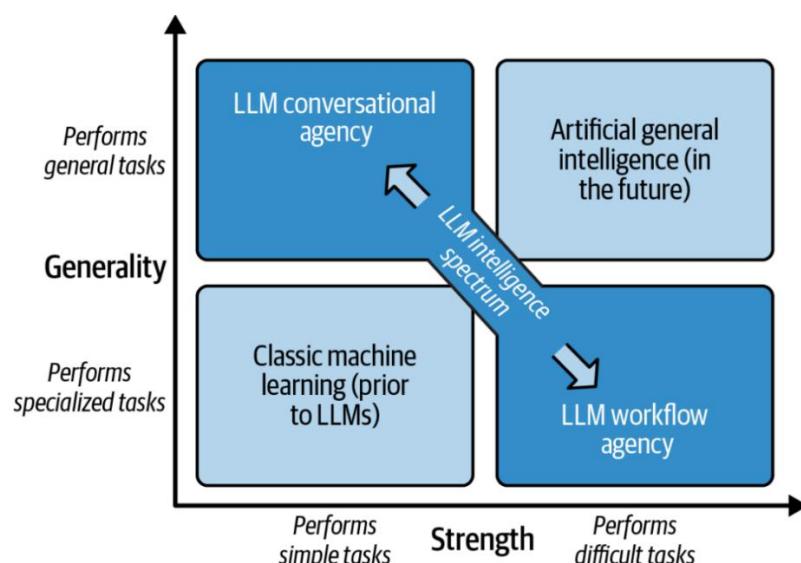


Abbildung 37: Anwendungsfelder von LLM-basierten Applikationen [83]

Tabelle 13: Vergleich zwischen LLM-Agenten und LLM-Workflows

	LLM-Konversationen	LLM-Workflows
Fokus	Interaktion & Kommunikation	Prozessoptimierung
Stärken	Allgemeine Aufgaben	Spezialisierte Aufgaben
Temperatur	Meist höher	Meist geringer
Beispiele	Chatbots, virtuelle Assistenten	Automatisierte Code-Optimierung

Da im Rahmen dieser Arbeit eine automatisierte Anwendung von Clean-Code-Prinzipien erfolgt, liegt der Fokus auf LLM-Workflows. Der typische Ablauf der Entwicklung einer solchen Applikation läuft folgendermaßen ab:

- 1. Definition des Ziels:**
→ Was soll der Workflow optimieren?
 - 2. Spezifizierung der Aufgaben**
→ Welche Aufgaben sind nötig, um das Ziel zu erreichen?
 - 3. Implementierung der Aufgaben**
→ Welche Prompts sind für die einzelnen Aufgaben nötig?
 - 4. Verbindung der Aufgaben**
→ In welcher Reihenfolge sollten die Aufgaben abgearbeitet werden?
- (5.) Optionale Optimierung des Workflows**
→ Gibt es Optimierungspotenzial bei den gewählten Prompts oder der Reihenfolge?

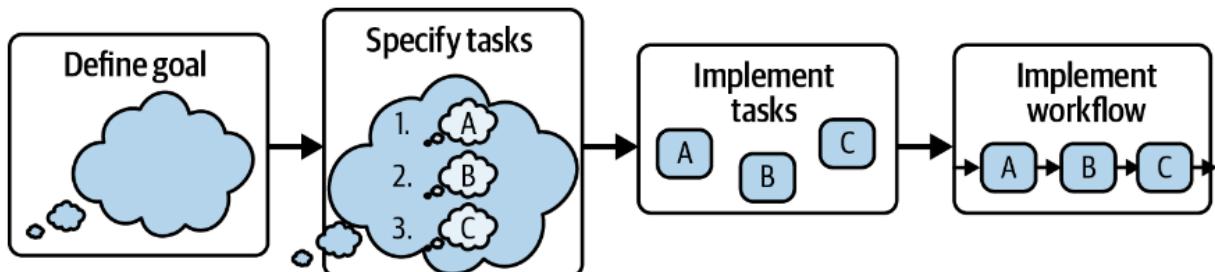


Abbildung 38: Ablauf der Entwicklung eines LLM-Workflows [83]

Dabei ist darauf zu achten, dass die Prompts der einzelnen Aufgaben folgende Charakteristika aufweisen [83]:

- **Modularität**
→ Prompt-Abschnitte sollten flexibel hinzugefügt und entfernt werden können.
- **Natürlichkeit**
→ Prompt-Abschnitte sollten sich nachvollziehbar in den Workflow bzw. in das Dokument implementieren lassen und dessen Format beibehalten. Bei Code-Vervollständigungen sollten Anweisungen nicht zwischen den Zeilen, sondern als Kommentar neben der respektiven Zeile angegeben werden.
- **Prägnanz**
→ Prägnantere Prompts, die denselben Kontext vermitteln, benötigen weniger Tokens und minimieren damit die Kosten.

2.8.1 Best Practices für effektive Prompts

Der Aufbau eines Prompts ist ein entscheidender Faktor für die Qualität der generierten Antwort eines LLMs. Bevor jedoch auf die optimale Prompt-Struktur eingegangen wird, ist es essenziell folgende Herausforderungen zu erläutern:

- **In-context learning [89]**
Desto näher sich eine Information am Ende des Prompts befindet, desto stärker beeinflusst sie das Verhalten des Modells.
- **The lost middle phenomenon [90]**
Während sich das Modell an Informationen am Anfang und Ende des Prompts einfach erinnern kann, hat es Schwierigkeiten mit Informationen in der Mitte.

Diese Effekte treten meist in der oberen Mitte des Prompts auf und können verstärkt bei langen Prompts beobachtet werden. Für diese Probleme existiert derzeit keine optimale Lösung, lediglich die Präzisierung des Prompts sowie eine strategische Anordnung der bereitgestellten Informationen können dazu beitragen, dieses Verhalten zu minimieren.

Darüber hinaus sind beim Umgang mit LLMs und infolgedessen bei der Verfassung von Prompts folgende Aspekte zu beachten [83]:

- *LLMs sind anfällig für irrelevante Informationen.*
- *LLMs müssen Prompts semantisch eindeutig interpretieren können.*
- *LLMs erfordern explizite und unmissverständliche Anweisungen.*
- *LLMs besitzen keine intrinsische Problemlösungsstrategie.*
- *LLMs führen keine selbstgesteuerten Denkprozesse aus.*

Auf Basis der erwähnten Verhaltensweisen ergeben sich folgende Anforderungen an Prompts:

- **Zuweisung einer Rolle [91][92]**
- **Einholen einer Expertenmeinung [92]**
- **Präzisierung der Anforderungen [91][92][94]**
- **Bereitstellung von Beispielen [91][93]**
- **Verwendung von Trennzeichen (z.B.: ``') [91][93]**
- **Festlegung des Ausgabeformats [92][93][94]**
- **Bei Bedarf: Spezifizierung der Arbeitsschritte [91][93]**
- **Bei Bedarf: Begrenzung der Wort-/Zeichenanzahl [91][93]**

Die Struktur eines Prompts hängt von der Art der Bedienung des LLMs ab:

- **User-Prompt**

Die Eingabe des User-Prompts erfolgt direkt über die bereitgestellte Benutzeroberfläche des Chatbots (z.B.: ChatGPT, Claude). Das Modellverhalten ändert sich dabei mit jeder neuen Anfrage.

Introduction	You are an Expert in establishing code quality in Python 3.10+.
Context	Your task is to ensure that the given code adheres to the following rules: <ul style="list-style-type: none"> • Follow PEP 8 and PEP 257 standards. • Ensure type hints are used appropriately.
Transition	Python-Code to apply rules to: <pre>```python def addNumbers(a,b): return a+b ``` </pre>
Refocus	<ul style="list-style-type: none"> • Give back the code in markdown format. • Do not include any explanation. • Modify only what is mentioned in the provided guidelines. • If the code is not compliant with the given guideline, you will be fined \$1000!

- **System-Prompt**

Die Eingabe des System-Prompts ist nur über spezielle Benutzeroberflächen (z.B.: OpenAI Playground, Anthropic Workbench) oder APIs möglich. Der Vorteil dieses Prompts ist, dass das Modellverhalten konstant bleibt und sich bei neuen Anfragen nicht verändert.

Role	# Role - You are an Expert in establishing code quality in Python (3.10+).
Context	# Context - Give back the code in markdown format. - Do not include any explanation. - Modify only what is mentioned in the provided guidelines. - If the code is not compliant with the given guideline, you will be fined \$1000!
Instructions	# Instructions - Your task is to ensure that the given code adheres to the following rules: - Follow PEP 8 and PEP 257 standards. - Ensure type hints are used appropriately.

Grundsätzlich kann zwischen Zero-Shot-Prompting und Few-Shot-Prompting unterschieden werden:

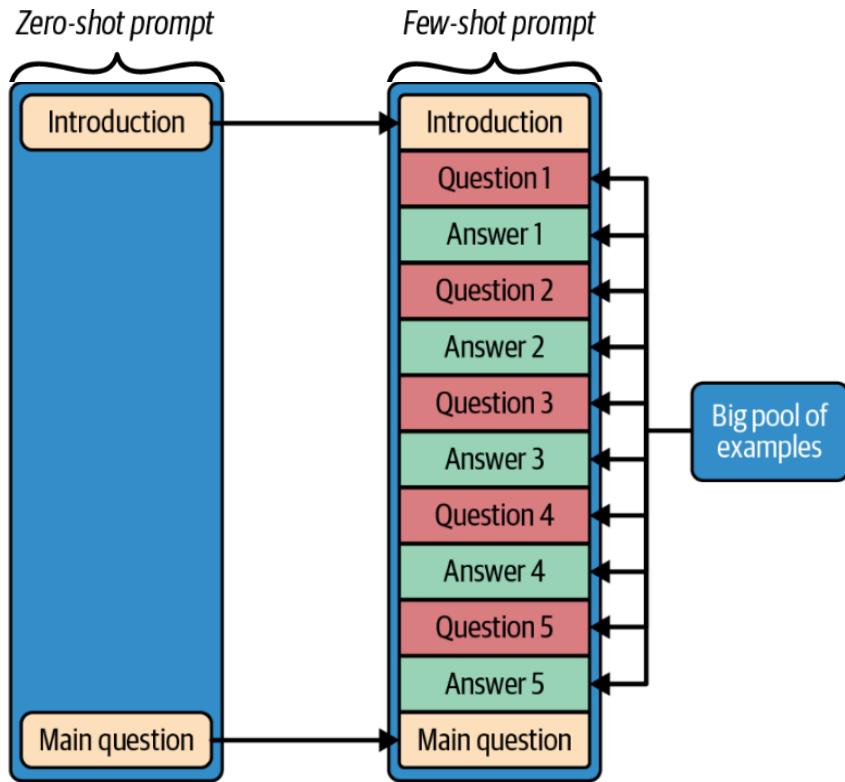


Abbildung 39: Vergleich der Struktur eines Zero-Shot-Prompts mit der eines Few-Shot-Prompts [83]

Wie sich diese Prompt-Techniken auf das Modellverhalten auswirken, hebt folgende Abbildung hervor:

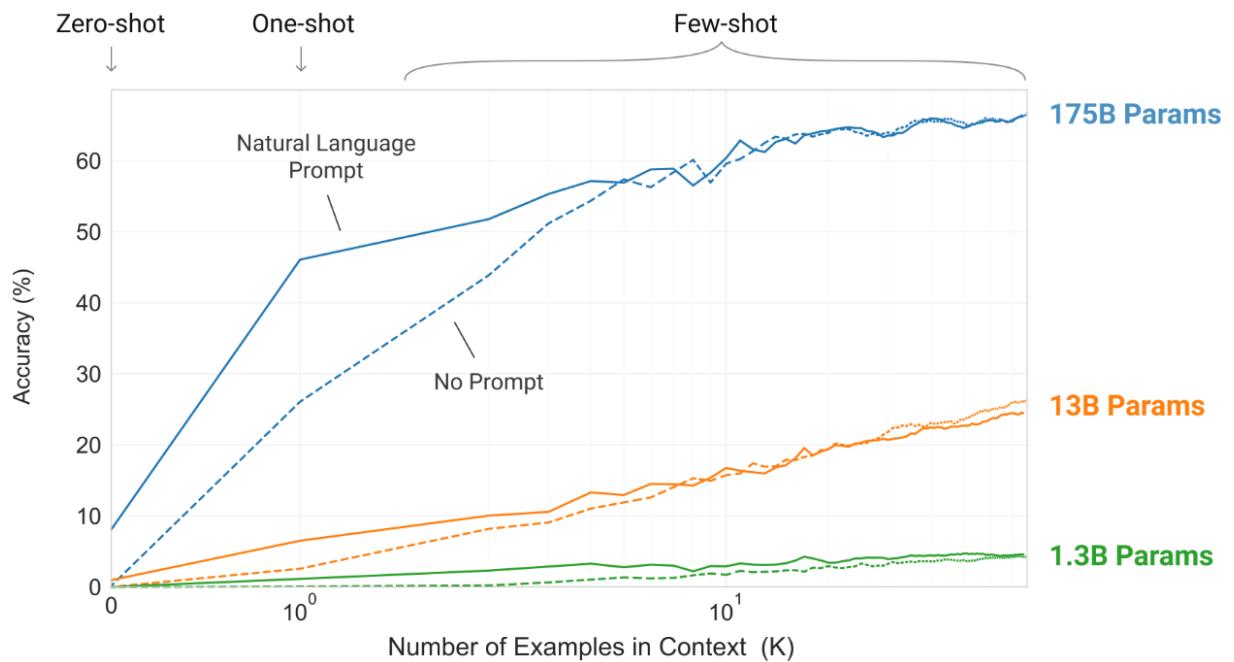


Abbildung 40: Einfluss von Modellgröße und Prompting auf die Genauigkeit bei der Entfernung von zufälligen Symbolen aus Wörtern [95]

Es zeigt sich, dass sowohl die Modellgröße als auch die Anzahl der angegebenen Beispiele die Genauigkeit erheblich verbessert. Insbesondere bei Modellen mit vielen Parametern scheint die zusätzliche Angabe einer sprachlichen Aufgabenbeschreibung die Leistung weiter zu steigern. Eine Auflistung weiterer Prompt Engineering Techniken sind unter [96] abrufbar.

2.8.2 Bedeutung des Eingabeformats

Das Format des Prompts, also ob Markdown, YAML, JSON oder Plain Text, hat einen Einfluss auf die Leistung eines LLMs. Auf Basis verschiedener Benchmarks zeigten sich Abweichungen in der Genauigkeit bei textbasierten Fragen, der Erfolgsrate bei Codegenerierung und der Qualität von Code-Übersetzungen. Während neuere bzw. größere Modelle wie GPT-4 weniger anfällig auf Formatänderungen sind, zeigen Vorgängermodelle wie GPT-3.5 eine deutliche Empfindlichkeit. Da für das GPT-4-Modell das Markdown-Format als optimales Eingabeformat empfohlen wird, wird angenommen, dass dies auch für die in dieser Arbeit verwendeten neueren und robusteren Modelle gilt. [94]

Prompts im Markdown-Format zu verfassen hat darüber hinaus weitere Vorteile [68]:

- **Verbesserte Interpretation durch umfangreiche Trainingsdaten:**
Da Markdown bereits im Jahre 2004 [98] entwickelt wurde und in der Vergangenheit eine der beliebtesten Auszeichnungssprachen (Markup languages) war [99], beispielsweise für die Verfassung von README.md-Dateien, befindet sich eine Vielzahl solcher Dateien im Internet. Da LLMs mit diesen Datensätzen trainiert werden, können sie Inhalte von Markdown-Dateien gut interpretieren und verarbeiten.
- **Benutzerfreundliche Erstellung**
Durch die einfache Syntax, gestaltet sich die Erstellung von Inhalten benutzerfreundlich und ermöglicht dem LLM, diese präzise zu erfassen.
- **Klare Struktur durch Formatierungselemente:**
Markdown-Dateien lassen sich mithilfe von Überschriften, Listen, Tabellen, Code-Blöcken und anderen Formatierungselementen in klar verständliche Abschnitte gliedern. Programmcodes können explizit mit dreifachen Backticks (```) vor und nach dem Block definiert werden. Wird neben den ersten Backticks auch die Programmiersprache angegeben (```python), so ermöglicht dies eine syntaxspezifische farbliche Hervorhebung. Dies hilft nicht nur dem Entwickler/der Entwicklerin den Programmcode einfacher nachzuvollziehen, sondern erlaubt es dem LLM, den Code effizienter zu analysieren.
- **Flexible Formatierung:**
Durch die klare Struktur von Markdown-Dateien, können Abschnitte flexibel angepasst und verschoben werden. Insbesondere bei langen Dokumenten ist dies sehr vorteilhaft.
- **Effiziente Referenzierung von Inhalten durch Hyperlinks:**
Ist eine Referenzierung zu einem anderen Abschnitt des Dokuments notwendig, so kann diese nachvollziehbar mit einem Hyperlink erfolgen. Diese Verlinkungen erlauben

nicht nur eine optimierte Navigation innerhalb des Dokuments, sondern auch eine bessere Erfassung der Zusammenhänge seitens des LLMs.

- **Einfache Darstellung von LLM-Outputs:**

Da Markdown leicht zu rendern ist, wird es häufig als direkte Ausgabe des LLM-Outputs verwendet.

Eine detaillierte Erläuterung sämtlicher Funktionen von Markdown ist [99] zu entnehmen.

3 Methodik

Um Quellcode optimieren zu können, ist es wichtig, die aktuellen zugrundeliegenden Praktiken zu verstehen. Während Quellcodeanalysen früher in persönlichen Peer-Reviews stattgefunden haben, wird heutzutage meist auf automatische LLM-Tools und Coding-Assistenten zurückgegriffen. Obwohl bereits einige Unternehmen den Einsatz von Tools, wie ChatGPT oder Github Copilot, dulden, fehlt eine klare Definition der zu bewertenden Parametern innerhalb des Unternehmens. Das Prompting erfolgt meist individuell auf Basis des jeweiligen Entwicklers und zeigt oftmals keinerlei Konsistenz. Ob dies dem Zeitdruck, einem Mangel an Kommunikation, einer unstrukturierten Arbeitsweise oder etwas Anderem geschuldet sei, soll im Rahmen dieser Arbeit nicht hinterfragt werden.

3.1 Clean-Code-Leitfaden für die Python-Entwicklung

Der Fokus dieser Arbeit liegt auf der Erarbeitung eines individuellen Leitfadens zur Optimierung der inneren Softwarequalität für die Programmiersprache Python. Der Leitfaden soll auf Basis des Feedbacks der Industrie (siehe Kapitel 2.4.10) und auf Basis der Hauptqualitätsmerkmale des zum Stand dieser Arbeit aktuellen ISO-Standards, der ISO/IEC 25010:2023, erstellt werden.

Obwohl die Umfrage aus dem Jahr 2012 ist und demnach etwas veraltet ist, bietet sie aufgrund der hochqualifizierten Teilnehmer – von welchen lediglich 15% weniger als fünf Jahre Arbeitserfahrung im Bereich der Softwareentwicklung hatten – einen aussagekräftigen Einblick in die damalige Qualitätssicherung von Softwaresystemen. Auf Basis der Qualität der Umfrage können Parallelen zu den heutigen ISO-Standards gezogen werden. Von den Befragten nutzen 28% die ISO 9126 direkt und 71,4% indirekt durch die Ergänzung mit unternehmensspezifischen Modellen. [39]

Da insbesondere Wert auf die Bedürfnisse der Entwickler gelegt wird und spezifische Clean-Code-Prinzipien in Python verwendet werden sollen, werden äußere Qualitätsmerkmale nicht berücksichtigt. Die Funktionalität und der Verwendungszweck eines Softwaresystems ist nämlich sehr individuell, weshalb für einige äußere Qualitätsmerkmale keine pauschale Bewertung getroffen oder Verbesserung erzielt werden kann.

Auf Basis der ISO 25010:2023 werden folgende innere Hauptqualitätsmerkmale fokussiert:

- **Maintainability** (Modularity, Reusability, Analysability, Modifiability, Testability)
- **Reliability** (Faultiness, Availability, Fault tolerance, Recoverability)

Der Fokus des Leitfadens liegt somit auf der Wartbarkeit und Zuverlässigkeit. Die Performance-Effizienz (Time behavior, Resource utilization, Capacity) wird dabei indirekt durch die Anwendung von pythonspezifischen Optimierungen, wie List Comprehensions, verbessert.

Bevor auf konkrete Clean-Code-Prinzipien eingegangen wird, soll die Gemeinsamkeit von namhaften Python-Projekten diskutiert werden:

Package	License	Line count	Docstrings (% of lines)	Comments (% of lines)	Blank lines (% of lines)	Average function length
HowDol	MIT	262	0%	6%	20%	13 lines of code
Diamond	MIT	6,021	21%	9%	16%	11 lines of code
Tablib	MIT	1,802	19%	4%	27%	8 lines of code
Requests	Apache 2.0	4,072	23%	8%	19%	10 lines of code
Flask	BSD 3-clause	10,163	7%	12%	11%	13 lines of code
Werkzeug	BSD 3-clause	25,822	25%	3%	13%	9 lines of code

Abbildung 41: Eigenschaften von namhaften Python-Projekten [68]

Folgendes kann festgestellt werden:

- Docstrings werden häufiger verwendet als Kommentare.
- Leerzeilen machen mit 11% bis 27% einen erheblichen Anteil des Codes aus. Dies ist vermutlich auf die Verwendung der PEP-8 Richtlinie zurückzuführen, welche unter anderem durch den bewussten Einsatz von Leerzeilen die Lesbarkeit verbessert.
- Obwohl die Codelänge bzw. die Zeilenanzahl stark variiert, befindet sich die durchschnittliche Funktionslänge mit 8 bis 13 Zeilen in einem ähnlichen Bereich.

Die Titel der Clean-Code-Prinzipien werden bewusst in englischer Sprache geschrieben, damit eine Zuweisung zu dem englischen Leitfaden erfolgen kann. Obwohl die Erklärung des Leitfadens im Rahmen dieser Arbeit in deutscher Sprache erfolgt, wird die zugrundeliegende englische Version im Anhang A: guideline.md bereitgestellt.

Beim Erstellen des Leitfadens wurden sowohl anerkannte Python-Richtlinien als auch relevante Fachliteratur herangezogen:

- *PEP-8* [68]
- *PEP-20* [101]
- *The Hitchhiker's Guide to Python: Best Practices for Development* [102]
- *The Art of Clean Code: Best Practices to eliminate complexity and simplify your life* [103]
- *Clean Code Principles And Patterns: Python Edition* [104]
- *Beyond the Basic Stuff with Python: Best Practices for writing Clean Code* [105]
- *PEP-484* [115]
- *PEP-257* [116]
- *PEP-282* [108]
- *PEP-287* [118]
- *Pandas docstring guide* [117]
- *Numpy docstring guide* [119][120]

3.1.1 General Pythonic Practices

Vereinfachen von for-Loops mit List Comprehensions

Durch die Verwendung von List Comprehensions wird nicht nur die Zeilenanzahl maßgeblich reduziert, sondern auch die Lesbarkeit verbessert. Darüber hinaus bietet die kompakte Notation oft eine performantere Lösung als die herkömmliche mehrzeilige Schleife. Dies trifft zumindest für das folgende Beispiel zu, da die List Comprehension interne Optimierungen besitzt während die for-Loop wiederholte append()-Aufrufe aufweist.

```
# WRONG
even_numbers = []
for number in numbers:
    if number % 2 == 0:
        even_numbers.append(number)

# CORRECT
even_numbers = [number for number in numbers if number % 2 == 0]
```

Nutzen des Context Managers beim Zugriff auf Dateien

Wird in Python mit Dateien, wie *.txt, *.json oder *.csv, gearbeitet, so ist es wichtig stets effizient mit den Ressourcen umzugehen. Wird eine Datei ohne Context-Manager (ohne with-Statement) manuell geöffnet und tritt beim Lesen des Inhalts ein Fehler auf, bleibt die Datei möglicherweise geöffnet, wodurch Ressourcenlecks entstehen können. Des Weiteren ist es empfehlenswert die Zeichenkodierung (beispielsweise UTF-8) anzugeben, um Formatierungsprobleme zu vermeiden.

```
# WRONG
input_file = open("input.txt", "r")
content = input_file.read()
input_file.close()

# BETTER
with open("input.txt", "r") as input_file:
    content = input_file.read()

# BEST
with open("input.txt", "r", encoding="utf-8") as input_file:
    content = input_file.read()
```

Einsatz von enumerate() statt range(len())

Der Einsatz der enumerate()-Funktion bietet eine simple Möglichkeit sowohl den Index als auch das zugehörige Element einer Iterable (z.B. einer Liste) abzurufen. Durch die direkte Zuweisung des Indexes werden Off-by-One-Fehler verhindert und dadurch das generelle Fehlerpotenzial minimiert. Darüber hinaus erlaubt die erleichterte Lesbarkeit der enumerate()-Funktion den Code prägnanter und wesentlich intuitiver zu gestalten.

```
# WRONG
for i in range(len(elements)):
    print(f"Index: {i}, Element: {elements[i]}")

# CORRECT
for i, element in enumerate(elements):
    print(f"Index: {i}, Element: {element}")
```

Verwenden von f-Strings anstelle von String Concatenation

Die String-Formatierung in Python hat bisher eine lange Entwicklungsgeschichte durchlaufen. Ursprünglich wurde auf den Einsatz des „+“-Operators zur Aneinanderkettung von Strings gesetzt. Diese Methode wies jedoch eine übermäßige Verwendung von Anführungs- und Plus-Zeichen auf, was nicht nur das Risiko von Formatierungsfehlern, insbesondere bei Abständen, erhöhte, sondern auch die Lesbarkeit massiv einschränkte.

Mit Python 2.6 wurde die `format()`-Methode etabliert, welche erstmals die Format Specification Mini-Language [110] nutzte. Dabei wurden geschwungene Klammern als Platzhalter verwendet, die dann mit der `format()`-Methode die jeweiligen Variablen und Formatierungsregeln anwendeten. Obwohl diese Methode übersichtlicher und besser lesbar ist als jene des „+“-Operators, gibt es mittlerweile eine noch elegantere und intuitivere Lösung.

Mit Python 3.6 wurden schließlich die beliebten f-Strings (kurz für Format-Strings) eingeführt. Sie werden mit einem `f` vor dem ersten Anführungszeichen gekennzeichnet und erlauben es, Variablen oder Operationen direkt innerhalb der geschwungenen Klammern anzugeben bzw. durchzuführen. Diese Lösung ist nicht nur die kompakteste, sondern ermöglicht auch einen optimalen Lesefluss, weshalb sie als bevorzugte Variante empfohlen wird.

Zur Erinnerung: Wie bereits mehrmals in dieser Arbeit erwähnt, empfiehlt die PEP-20: „The Zen of Python“ [101] mit der Regel „*There should be one-- and preferably only one --obvious way to do it.*“ einen Ansatz konsistent zu verwenden und nicht zwischen verschiedenen Methoden zu wechseln.

```
# WRONG
print(name + " is " + str(age) + " years old.")

# BETTER
print("{} is {} years old.".format(name, age))

# BEST
print(f"{name} is {age} years old.")
```

Ersetzen von Print-Statements mit Logging

Mit der Veröffentlichung von Python 2.3 wurde im Rahmen der PEP-282 [108] ein Logging-Modul als eingebautes Package in die Standardbibliothek eingeführt. Während print-Statements ebenfalls ihr Daseinsberechtigung besitzen, sollten sie ausschließlich für einfache bzw. temporäre Debugging-Zwecke verwendet werden (*Common code smell: Print Debugging* nach [102] und [105]). Logging hingegen ist ein Grundbestandteil von produktionsreifem Code, welcher einerseits zu diagnostischen und andererseits zu analytischen Zwecken eingesetzt wird:

- **Diagnostic logging**

Hierbei handelt es sich um Ereignisse, die mit dem Betrieb der Software in Zusammenhang stehen. Meldet ein Benutzer beispielsweise einen Fehler, so kann der verantwortliche Entwickler die Logs heranziehen, um die Ursache zu identifizieren.

- **Audit logging**

Umschreibt jene Ereignisse, die für geschäftliche Analysezwecke verwendet werden.

Dabei handelt es sich unter anderem um Nutzerinteraktionen, wie Klickverläufe, die mit weiteren Daten, wie getätigten Einkäufen, kombiniert werden, um das Nutzerverhalten in Reports zu analysieren und in weiterer Linie Geschäftsprozesse gezielt zu optimieren.

Wichtig ist jedoch zu beachten, dass folgende Informationen nicht geloggt werden sollten [104]:

- Personenbezogene Informationen
- Gesetzlich verbotene Informationen
- Benutzerdaten, deren Erfassung der Nutzer nicht zugestimmt hat
- Passwörter
- Session-IDs
- Access tokens
- Encryption keys
- Database connection strings

Darüber hinaus beinhalten die mit dem Log-Event erstellen Log-Records zusätzliche relevante Daten, wie den Dateinamen, das Dateiverzeichnis, die Funktion und die Zeilennummer.

Anmerkung: Welches Log-Level eingesetzt werden soll, kann dem OWASP Application Logging Vocabulary Cheat Sheet [109] entnommen werden. Um den Rahmen dieser Arbeit aber nicht zu sprengen, wird der Einsatz des korrekten Log-Levels im Produktionsumfeld lediglich am Rande erwähnt.

```
# WRONG
def divide(dividend, divisor):
    print(f"Received input: dividend={dividend}, divisor={divisor}")

    if divisor == 0:
        print("Error: Division by zero attempted!")
        raise ValueError("Division by zero is not allowed.")

    result = dividend / divisor
    print(f"Calculation successful: {dividend} / {divisor} = {result}")
    return result

# CORRECT
import logging

logging.basicConfig(
    level=logging.INFO,
    format=(
        "[%(asctime)s.%03d][%(levelname)s]"
        "[%(filename)s:%(lineno)d - %(funcName)s()]: %(message)s"
    ),
    datefmt="%d-%m-%Y %H:%M:%S",
)
```

```
)  
  
def divide(dividend, divisor):  
    logging.info(f"Received input: {dividend=}, {divisor=}")  
  
    if divisor == 0:  
        logging.error("Error: Division by zero attempted!")  
        raise ValueError("Division by zero is not allowed.")  
  
    result = dividend / divisor  
    logging.info(f"Calculation successful: {dividend} / {divisor} = {result}")  
    return result
```

3.1.2 Refactoring

Vermeiden von Verschachtelungen durch Verwendung von max. 3 Einrückungsebenen

Die PEP-20 [101] hebt mit der Regel „*Flat is better than nested.*“ hervor, dass der Einsatz einer geringen Einrückungstiefe die Nachvollziehbarkeit, Lesbarkeit und Wartbarkeit positiv beeinflusst. Obwohl eine geringere Einrückungstiefe eine Verlängerung der Zeilenlänge bewirkt, überwiegen die Vorteile der optimierten Code-Struktur und der verbesserten Übersichtlichkeit (*Don't Use Too Many Levels of Indentation* nach [103]). Grundsätzlich obliegt es dem/der Entwickler:in, auf welche Anzahl die Einrückungstiefe begrenzt wird. Im Rahmen dieser Arbeit wird auf Erfahrungswerte zurückgegriffen und die maximale Tiefe auf drei Ebenen begrenzt.

```
# WRONG
def check_permissions(user):
    if user:
        if user.is_valid():
            if user.has_role("admin"):
                if user.has_permission("edit"):
                    print("Access granted!")
                else:
                    print("Permission denied!")
            else:
                print("User has no admin role!")
        else:
            print("User is invalid!")
    else:
        print("No user provided!")
```

```
# CORRECT
def check_permissions(user):
    if not user:
        print("No user provided!")
        return

    if not user.is_valid():
        print("User is invalid!")
        return

    if not user.has_role("admin"):
        print("User has no admin role!")
        return

    if not user.has_permission("edit"):
        print("Permission denied!")
        return

    print("Access granted!")
```

Aufteilung langer Funktionen in kleinere mit maximal 10 Zeilen (Small Is Beautiful & Single Responsibility Principle)

Ähnlich wie verschachtelte Funktionen/Methoden, verschlechtert die Verwendung von Funktionen mit einer hohen Zeilenanzahl die Nachvollziehbarkeit, Lesbarkeit und Wartbarkeit fundamental. Darüber hinaus ermöglicht eine kurze, prägnante Funktion/Methode bei Auftritt von unbekannten Fehlern einen optimierten Debugging-Prozess (*Small Is Beautiful* nach [103]).

Des Weiteren sollte jede Funktion/Methode grundsätzlich nur eine Verantwortlichkeit besitzen (*Single Responsibility Principle* nach [103] und [104]).

Bevor die Definition der maximalen Zeilenanzahl erfolgt, sollen an dieser Stelle kurz die Nachteile der Verwendung von vielen Funktionen diskutiert werden: [105]

- Sehr kurze Funktionen erhöhen die Gesamtanzahl an Funktionen.
- Eine höhere Anzahl an Funktionen gestaltet den Programmcode komplizierter.
- Die Beziehung zwischen den Funktionen wird komplexer.
- Der Aufwand zielführende Benennungen zu finden ist höher.
- Der Dokumentationsaufwand, z.B. durch Docstrings, steigt.

Aufgrund dieser Nachteile ist es essenziell, ein Gleichgewicht zwischen kurzen, gut strukturierten Funktionen und einer übermäßigen Anzahl an Funktionen zu finden. Die Begrenzung auf 10 Zeilen pro Funktion/Methode basiert daher auf der durchschnittlichen Funktionslänge von namhaften Python-Projekten (siehe Abbildung 41).

```
# WRONG
def process_order(order):
    total = sum(item["price"] * item["quantity"] for item in order["items"])

    if total > 100:
        discount = total * 0.1
    else:
        discount = 0

    final_price = total - discount

    print(f"Final Price: {final_price}")

# CORRECT
def calculate_total(order):
    return sum(item["price"] * item["quantity"] for item in order["items"])

def apply_discount(total):
    return total * 0.1 if total > 100 else 0

def process_order(order):
    total = calculate_total(order)
    discount = apply_discount(total)
    final_price = total - discount
    print(f"Final Price: {final_price}")
```

Entfernen von dupliziertem Code

Eine der fundamentalsten Refactoring-Maßnahmen ist das Entfernen von dupliziertem Code, welcher durch das Kopieren und Einfügen von Code-Abschnitten an mehreren Stellen eines Programms entstehen kann. Das Problem bei dupliziertem Code ist, dass eine Änderung meist aufwendig ist und bei unzureichender Anpassung von allen Abschnitten ein unerwartetes Verhalten bzw. Fehler entstehen können. Die Lösung ist die Deduplikation des Codes durch Anwendung des DRY-Prinzips („Don't Repeat Yourself“), um die wiederholende Logik abzukapseln. Dadurch wird es ermöglicht, dass der Code nur an einer Stelle angepasst werden muss, um das gesamte Verhalten bequem anpassen zu können (*Common code smell: Duplicate Code* nach [104] und [105]). Desto häufiger ähnliche Abschnitte auftreten, desto dringlicher sollte eine Deduplikation erfolgen.

```
# WRONG
print("Good morning!")
print("How's your mood right now?")
mood = input()
print(f"Glad to hear that you feel {mood}.")

print("Good afternoon!")
print("How's your mood right now?")
mood = input()
print(f"Glad to hear that you feel {mood}.")

print("Good evening!")
print("How's your mood right now?")
mood = input()
print(f"Glad to hear that you feel {mood}.")
```

```
# BETTER
def check_mood():
    print("How's your mood right now?")
    mood = input()
    print(f"Glad to hear that you feel {mood}.")

print("Good morning!")
check_mood()

print("Good afternoon!")
check_mood()

print("Good evening!")
check_mood()
```

```
# BEST
def check_mood(time_of_day):
    print(f"Good {time_of_day}!")
    print("How's your mood right now?")
    mood = input()
    print(f"Glad to hear that you feel {mood}.")

times_of_day = ["morning", "afternoon", "evening"]
for time_of_day in times_of_day:
    check_mood(time_of_day)
```

Ein Statement pro Zeile

Die PEP-20 [101] untermauert mit der Regel „Sparse is better than dense.“, dass der Code lesbarer ist, wenn pro Zeile maximal ein Statement definiert wird. Unabhängige Anweisungen sollten daher stets voneinander getrennt werden, darunter fällt unter anderem die Definition und Abfrage einer Variable. Insbesondere bei der Verwendung eines Versionskontrollsysteams, wie beispielsweise Git, erleichtert dieser Ansatz die Nachvollziehbarkeit von Änderungen drastisch. Obwohl die Zeilenanzahl des Codes erhöht wird, verbessert sich die Lesbarkeit bzw. die Wartbarkeit maßgeblich, da jede Änderung einer expliziten Zeile zugeordnet werden kann.

Hinweis: Obwohl List Comprehensions aufgrund ihrer Prägnanz hiervon ausgenommen sind, sollten auf deren Übersichtlichkeit geachtet werden.

```
# WRONG
customer_name = "Albert Einstein", customer_id = 1234
```

```
# CORRECT
customer_name = "Albert Einstein"
customer_id = 1234
```

```
# WRONG
if number > 0 and number % 2 == 0 and number % 99 == 0:
    print(f"{number} is valid!")
```

```
# CORRECT
is_positive = number > 0
is_even = number % 2 == 0
is_divisible_by_ninetynine = number % 99 == 0

if is_positive and is_even and is_divisible_by_ninetynine:
    print(f"{number} is valid!")
```

Entfernen von auskommentiertem “toten” Code

Zu Debugging-Zwecken ist es durchaus akzeptabel, bestimmte Programmzeilen auszukommentieren, um den Code schrittweise zu analysieren und eine gezielte Fehlersuche zu gewährleisten. Durch die temporäre Deaktivierung einzelner Codeabschnitte lassen sich potenzielle Fehlerquellen eingrenzen, ohne dabei den zugrunde liegenden Programmcode fundamental ändern zu müssen. In der Praxis passiert es jedoch häufig, dass Entwickler vergessen das Kommentarzeichen (#) nach dem Abschluss des Debugging-Prozesses zu entfernen, weshalb es für andere Entwickler:innen schwierig sein kann, den Code zu verstehen. Neben dem klassischen Kommentarzeichen werden auch Single-Line ('...', "...') oder Multi-Line Strings ("...", """...""") verwendet (*Common code smell: Commented-Out Code and Dead Code* nach [105]). Daher sollten veraltete Programmzeilen entfernt werden, da diese ohnehin durch Versionskontrollsysteme, wie Git, dokumentiert werden. Wichtig ist es jedoch zu beachten, dass irrtümlich keine Docstrings entfernt werden (dazu in Kapitel 3.1.7 mehr).

```
# WRONG
def calculate_total(price, discount):
    # total = price + (price * discount)
    return price * (1 + discount)
```

```

# WRONG
def calculate_total(price, discount):
    'total = price + (price * discount)'
    return price * (1 + discount)

# WRONG
def calculate_total(price, discount):
    "total = price + (price * discount)"
    return price * (1 + discount)

# WRONG
def calculate_total(price, discount):
    ...
    total = price + (price * discount)
    ...
    return price * (1 + discount)

# WRONG
def calculate_total(price, discount):
    """
    total = price + (price * discount)
    """
    return price * (1 + discount)

# CORRECT
def calculate_total(price, discount):
    return price * (1 - discount)

```

Globale Variablen sollten vermieden werden

Funktionen bzw. Methoden können als eigenständige Programme innerhalb eines größeren Programms angesehen werden. Sie beinhalten lokale Variablen, die nach erfolgreicher Ausführ wieder entfernt werden. Das bedeutet, dass eine Funktion bzw. Methode in sich geschlossen ist und ausschließlich von den jeweiligen Argumenten abhängt. Werden hingegen globale Variablen innerhalb einer Funktion bzw. Methode verwendet, so geht diese Isolation durch die implizite Definition des globalen Eingabewerts verloren. Tritt nun ein Fehler in einem bestimmten Codeabschnitt auf, so kann dies daran liegen, dass die verwendete globale Variable an einer völlig anderen Stelle adaptiert wurden. Während dies in Programmen mit geringer Zeilenanzahl kaum einen signifikanten Einfluss hat, kann dies in umfangreichen Programmen zu intransparenten Fehlermeldungen führen, wodurch eine zielgerichtete Fehleranalyse erschwert, und der Wartungsaufwand erhöht wird. Aus diesem Grund sollte die Nutzung von globalen Variablen auf ein Minimum reduziert werden (*Limit the use of global variables* nach [105]).

```

# WRONG
def read_api_key() -> None:
    load_dotenv	override=True)
    global API_KEY
    API_KEY = os.getenv("API_KEY")

# CORRECT
def read_api_key() -> str:
    load_dotenv	override=True)
    API_KEY = os.getenv("API_KEY")
    return API_KEY

```

Entfernen von redundanten Kommentaren

Obwohl es teilweise hilfreich sein kann ergänzende Kommentare bereitzustellen, können redundante Kommentare Missverständnisse verursachen. Wird darauf vergessen neben dem Source-Code auch die dementsprechenden Kommentare anzupassen, so kann dies zu Widersprüchen zwischen Code und Kommentaren führen, wodurch die Nachvollziehbarkeit vermindert wird. Während der Source-Code stets das Verhalten des Programmes widerspiegelt, bieten Kommentare lediglich eine optionale ergänzende Dokumentation. Aus genannten Gründen sollten Kommentare stets mit Vorsicht eingesetzt werden. Doch, ob ein Kommentar, insbesondere von einem anderen Verfasser, entfernt werden kann, ist häufig nicht pauschal beantwortbar. Worauf aber in den meisten Fällen verzichtet werden kann, sind redundante Kommentare, die ohnehin durch den Code selbst ersichtlich sind. Sollte der Code ohne Kommentare nicht verständlich genug sein, dann deutet dies darauf hin, dass der Code in Hinsicht auf Benennung und Struktur optimiert werden sollte. (*Clean Code Principle: Avoid Unnecessary Comments* nach [103] und [104]).

```
# WRONG
original_price = 1000 # Original price of the product
discount_rate = 0.1 # Discount rate (0.1 = 10%)

# Calculate the discounted price for the current year
discounted_price = original_price * (1 - discount_rate)

# Print the discounted price
print(f"{discounted_price}$")

# CORRECT
original_price = 1000
discount_rate = 0.1

discounted_price = original_price * (1 - discount_rate)
print(f"{discounted_price}$")
```

Für die Dokumentation des Verhaltens von Modulen, Klassen, Methoden oder Funktionen ist es wesentlich zielführender einen aussagekräftigen Docstring zu verwenden (dazu in Kapitel 3.1.7 mehr).

3.1.3 Main Function and Main Name Idiom

Main-Funktion

Zur besseren Nachvollziehbarkeit, welcher Teil des Codes tatsächlich ausgeführt werden soll, empfiehlt es sich eine *main()*-Funktion zu definieren. Innerhalb dieser Hauptfunktion wird die Logik des Programmes definiert, um die Struktur des Programmcodes zu verbessern. Teilweise passiert es nämlich, dass fälschlicherweise an einer beliebigen Stelle im Programmcode Variablen definiert, Funktionen aufgerufen oder Klassen instanziert werden, wodurch die Lesbarkeit und infolgedessen die Wartbarkeit und Erweiterbarkeit stark eingeschränkt werden. Die Verwendung einer *main()*-Funktion erzwingt daher eine unumgängliche Restrukturierung der Logik des Programmcodes. Dadurch wird anderen Entwicklern schnell und einfach vermittelt, in welchem Abschnitt der Ausgangspunkt für ein zielgerichtetes Debugging erfolgen kann. Wie in einem späteren Kapitel beschrieben, sollte ebenfalls ein Return-Type, in diesem Fall *None*, deklariert werden.

```
# WRONG
print("Hello World!")

# BETTER
def main() -> None:
    print("Hello World!")

main()
```

Des Weiteren sollte die ***main()*-Funktion** lediglich die ***Logik des Programmcodes*** beinhalten. Jeder Schritt innerhalb dieser Logik sollte deswegen in eine separate Funktion ausgelagert werden. Dies hilft einerseits dabei, den Programmcode lesbarer zu gestalten und andererseits dabei die geschriebenen Klassen, Methoden und Funktionen wiederverwendbar und modular zu halten.

```
# WRONG
def square(number):
    return number**2

def save_result(number, squared):
    with open("results.txt", "w") as file:
        file.write(f"The square of {number} is {squared}\n")
    print("Result saved to results.txt")

def main():
    while True:
        try:
            number = int(input("Enter a number to be squared: "))
            break
        except ValueError:
            print("Invalid input. Please enter a valid number (int).")
    squared = square(number)
    save_result(number, squared)
```

```

# CORRECT
def get_number():
    while True:
        try:
            return int(input("Enter a number to be squared: "))
        except ValueError:
            print("Invalid input. Please enter a valid number (int).")

def square(number):
    return number**2

def save_result(number, squared):
    with open("results.txt", "w") as file:
        file.write(f"The square of {number} is {squared}\n")
    print("Result saved to results.txt")

def main():
    number = get_number()
    squared = square(number)

    save_result(number, squared)

```

Darüber hinaus sollte die ***main()*-Funktion** aus Formatierungsgründen immer die ***letzte Funktion*** innerhalb eines Programms sein.

```

# WRONG
def main() -> None:
    ...

def some_function() -> None:
    ...

# CORRECT
def some_function() -> None:
    ...

def main() -> None:
    ...

```

Main-Name-Idiom

Wie bereits mehrmals erwähnt wurde, ist die Wiederverwendbarkeit des zugrundeliegenden Programmcodes eine Qualitätseigenschaft eines Softwaresystems. Würde lediglich eine *main()*-Funktion, wie im vorherigen Abschnitt beschrieben, deklariert und aufgerufen werden, dann würde diese beim Import des Python-Skripts, um auf die darin befindlichen Klassen oder Funktionen zuzugreifen, ausgeführt werden. Dies kann zu einem unerwarteten Verhalten führen, wenn die ursprüngliche Datei nur für eine direkte Ausführung gedacht war. Dieses Problem wird durch den Einsatz eines Main-Name-Idioms (*if __name__ == "__main__":*) gelöst. Ausschließlich, wenn das Python-Skript direkt ausgeführt wird, wird die Logik innerhalb

des Main-Name-Idioms gestartet. Wenn das Python-Skript hingegen in ein anderes Python-Skript importiert wird, dann geschieht dies nicht.

```
# CORRECT
def main() -> None:
    print("Hello World!")

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

Da die PEP-8 [68] bzw. die PEP-257 [118] definitionsgemäß besagt, dass alle Funktionen einen Docstring besitzen sollten, wird dies von Lintern, beispielweise von pylint, als „missing-function-docstring“-Fehler erkannt. Obwohl die Python Community darüber debattiert, ob die reine Exekution des Codes in Form der *main()*-Funktion einen Docstring benötigt, hilft es unerfahrenen Entwicklern dabei, das Prinzip dahinter besser zu verstehen.

```
# BETTER
def main() -> None:
    """Executes the main functionality of the script."""
    print("Hello World!")

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

Zusammengefasst kann gesagt werden, dass die Kombination aus der Definition einer *main()*-Funktion und der Verwendung eben dieser innerhalb eines Main-Name-Idioms nicht nur zu einer besseren Lesbarkeit beiträgt, sondern auch verhindert, dass beim Import eines Python-Skripts in eine andere, ein ungewollter Codeabschnitt ausgeführt wird.

3.1.4 Naming Conventions

Allgemeine Benennungsregeln

Verwendung der englischen Sprache bestehend aus ASCII-Zeichen (keine Umlaute)

Während diese Regel für erfahrene Softwareentwickler selbstverständlich ist, beginnen Anfänger oder Quereinsteiger häufig in ihrer Muttersprache zu programmieren. Dies sollte strengstens vermieden werden, da für eine kollaborative Zusammenarbeit im internationalen Umfeld die Verwendung der englischen Sprache eine Grundvoraussetzung ist. Darüber hinaus kann der Einsatz von nicht ASCII-Zeichen, wie zum Beispiel von Umlauten, teils schwer erkennbare Encoding-Fehler verursachen.

```
# WRONG
öffnungszeiten = {...}
straße = "..."
```



```
# CORRECT
opening_hours = {...}
street = "..."
```

Nutzung von kurzen und geläufigen Begriffen

Die Benennung von jeglichen Variablen, Funktionen, Methoden, Klassen und Objekten ist so zu wählen, dass sie einerseits prägnant und andererseits geläufig ist. Kurze und einfache Namen tragen zu einer besseren Lesbar- bzw. Wartbarkeit bei und erlauben dadurch eine schnellere Einarbeitung in fremden Code. Insbesondere bei Funktionen und Methoden erfordert die Benennung eine hohe Aufmerksamkeit, da die verwendeten Verben für Unklarheit sorgen können. Neben dem unten angeführten Beispiel, werden geläufige Verben, wie *add*, *create*, *read*, *save*, *set* oder *update*, gerne eingesetzt. [111]

```
# WRONG
def retrieve_data():
    ...
# CORRECT
def get_data():
    ...
```

Wähle einen Begriff und verwende ihn konsistent

Die Qualität eines Programmcodes wird unter anderem durch die Konsistenz der Namensgebung beeinflusst. Wird für ähnliche Konzepte ein anderer Begriff verwendet, so verursacht dies einen erhöhten Aufwand sich im Code zurecht zu finden. Durch eine einheitliche Begriffswahl und dessen konsequenter Anwendung, entsteht ein sauber strukturiertes und verständliches System. Optional kann, insbesondere bei großen Teams, die Definition eines Wörterverzeichnisses hilfreich sein.

```
# WRONG
def get_data():
    ...
```

```
def process_info():
    ...
```

```
def save_content():
    ...
```

```
# CORRECT
def get_data():
    ...
```

```
def process_data():
    ...
```

```
def save_data():
    ...
```

Vermeidung von ungewöhnlichen Abkürzungen

Um die Lesbarkeit und Verständlichkeit eines Programmcodes nicht negativ zu beeinflussen, sollten ungewöhnliche Abkürzungen unbedingt vermieden werden. In Extremfällen, wenn die Benennung einer Komponente sehr lang ist und dadurch den Lesefluss maßgeblich stört, darf sie grundsätzlich durch eine sinnvolle, geläufige und verständliche Abkürzung ersetzt werden (z.B.: *very_very_long_string* → *very_very_long_str*). Angemerkt sei hier jedoch, dass jede Abkürzung einen nicht notwendigen Interpretationsrahmen ermöglicht.

```
# WRONG
cstmr = [...]
nbrs = [...]
temp_var = {...}
```

```
# CORRECT
customers = [...]
numbers = [...]
temperature_variance = {...}
```

Verwendung von aussagekräftigen Namen, um die Funktionalität zu verdeutlichen

Insbesondere bei Funktionen und Methoden ist es essenziell, die Benennung so zu wählen, dass sie für fremde Entwickler nachvollziehbar ist. Die Funktionalität sollte idealerweise direkt aus der Benennung abgeleitet werden und ohne den Einsatz von Kommentaren oder Debugging auskommen. Des Weiteren bietet ein aussagekräftiger Name eine verbesserte Lesbarkeit.

```
# WRONG
def convert():
    ...
```

```
# CORRECT
def eur_to_usd():
    ...
```

Vermeidung von zweideutigen Namen durch Angabe eines Kontexts

Die Vermeidung von zweideutigen Benennungen ist in der Praxis oftmals nicht einfach, da Entwickler unterbewusst durch ihre individuellen Annahmen, Perspektiven oder Erfahrungswerte beeinflusst werden. Ein Name, der in den Augen des Verfassers völlig offensichtlich ist, erscheint einem fremden Entwickler möglicherweise fragwürdig. In dem angeführten Beispiel, geht beispielweise nicht heraus, welcher Dollar gemeint ist - der amerikanische, der kanadische oder doch vielleicht ein anderer? Oft hilft es sich selbst folgende Frage zu stellen: „Würde jemand ohne meinen Kontext oder mein Domänenwissen die Funktionalität allein durch den Namen verstehen?“ Teilweise bietet sich auch die Kombination aus Verb und Substantiv an, wie zum Beispiel `get_user_profile(...)`. Desto expliziter der Name, desto besser.

```
# WRONG
def euro_to_dollar():
    ...
# CORRECT
def eur_to_usd():
    ...
```

Grundsätzliche Vermeidung der Definition von Datentypen im Variablenamen

Grundsätzlich gilt, dass der Name einer Variable bereits ihren Zweck beschreiben sollte. Das Anhängen des Datentyps am Ende des Namens würde hier zu einer unnötigen Redundanz führen. Durch den Einsatz von Type Hints lässt sich der Datentyp einer Variable schnell eruieren – dazu aber im nächsten Kapitel mehr.

```
# WRONG
customer_list = [...]
country_to_currency_dict = {...}
PI_VALUE = 3.14
# CORRECT
customers = [...]
country_to_currency = {...}
PI = 3.14
```

Es gibt jedoch Ausnahmefälle, in welchen die Angabe des Datentyps durchaus sinnvoll ist. Wird eine Variable nämlich in verschiedenen Formaten verwendet oder in andere Datentypen konvertiert, darf der Datentyp aus Gründen der verbesserten Klarheit angehängt werden. Häufig verwendete Bezeichnungen sind: `_str`, `_int`, `_float`, `_list`, `_dict`, `_set`.

```
# CORRECT
age_int = 33
age_str = str(age_int)
```

Eingebaute Namen sollten nicht überschrieben werden

Die Erweiterung eines Unterstrichs `_` am Ende eines Namens ist die empfohlene Vorgehensweise, um eine Kollision mit eingebauten Komponenten zu vermeiden. Beispielweise ist der Einsatz eines Unterstrichs, wie `class_`, einer unverständlichen Abkürzung, wie `clss`, vorzuziehen.

```
# WRONG
class Order:
    def __init__(self, id):
        self.id = id

# CORRECT
class Order:
    def __init__(self, id_):
        self.id_ = id_
```

Folgende Wörter sind reserviert und sollten daher nicht überschrieben werden:

ArithmeticError, AssertionError, AttributeError, BaseException, BaseExceptionGroup, BlockingIOError, BrokenPipeError, BufferError, BytesWarning, ChildProcessError, ConnectionAbortedError, ConnectionError, ConnectionRefusedError, ConnectionResetError, DeprecationWarning, EOFError, Ellipsis, EncodingWarning, EnvironmentError, Exception, ExceptionGroup, False, FileExistsError, FileNotFoundError, FloatingPointError, FutureWarning, GeneratorExit, IOError, ImportError, ImportWarning, IndentationError, IndexError, InterruptedError, IsADirectoryError, KeyError, KeyboardInterrupt, LookupError, MemoryError, ModuleNotFoundError, NameError, None, NotADirectoryError, NotImplemented, NotImplementedError, OSError, OverflowError, PendingDeprecationWarning, PermissionError, ProcessLookupError, RecursionError, ReferenceError, ResourceWarning, RuntimeError, RuntimeWarning, StopAsyncIteration, StopIteration, SyntaxError, SyntaxWarning, SystemError, SystemExit, TabError, TimeoutError, True, TypeError, UnboundLocalError, UnicodeDecodeError, UnicodeEncodeError, UnicodeError, UnicodeTranslateError, UnicodeWarning, UserWarning, ValueError, Warning, ZeroDivisionError, __build_class__, __debug__, __doc__, __import__, __loader__, __name__, __package__, __spec__, abs, aiter, all, and, anext, any, as, ascii, assert, async, await, bin, bool, break, breakpoint, bytearray, bytes, callable, chr, class, classmethod, compile, complex, continue, copyright, credits, def, del, delattr, dict, dir, divmod, elif, else, enumerate, eval, except, exec, exit, filter, finally, float, for, format, from, frozenset, getattr, global, globals, hasattr, hash, help, hex, id, if, import, in, input, int, is, isinstance, issubclass, iter, lambda, len, license, list, locals, map, max, memoryview, min, next, nonlocal, not, object, oct, open, or, ord, pass, pow, print, property, quit, raise, range, repr, return, reversed, round, set, setattr, slice, sorted, staticmethod, str, sum, super, try, tuple, type, vars, while, with, yield, zip

Diese Liste sollte laufend erweitert werden, um neue reservierte Wörter zu berücksichtigen, die mit neuen Python-Versionen hinzugefügt werden.

Weitere relevante datentypspezifische Benennungsregeln

Benennung von Modulen

Die Benennung von Modulen sollte laut PEP-8 [68] prägnant sein und dem lower snake_case entsprechen. Das heißt alle Buchstaben werden kleingeschrieben und Abstände durch Underscores „_“ ersetzt.

```
# WRONG
import SomeModule
import someModule
import Somemodule
import somemodeule
import Some_Module
import some_Module
import Some_module
```

```
# CORRECT
import module
import some_module
```

Benennung von Konstanten

Der Einsatz von “Magic numbers”, also von fest codierten numerischen Variablen direkt im Programmcode ist zu vermeiden. Ohne Kontext geht der Zweck und die Bedeutung der Konstante nicht eindeutig heraus. Womöglich handelt es sich um eine fundamentale Konstante, die das Verhalten oder das Ergebnis des Programms maßgeblich beeinflusst. Mit steigender Komplexität bzw. Codelänge wird die Nachvollziehbarkeit problematisch. Werden beispielweise im selben Programmcode mehrere „Magic numbers“ eingesetzt, die durch Zufall denselben numerischen Wert besitzen, aber völlig andere Bedeutungen und Zwecke erfüllen, so ist das Debugging für fremde Entwickler unmöglich.

Damit solche Missverständnisse vermieden werden und die Lesbarkeit verbessert wird, sollten Konstanten stets dem upper SNAKE_CASE entsprechen. Das bedeutet, dass alle Buchstaben großgeschrieben werden und Abstände mit Underscores „_“ symbolisiert werden.

```
# WRONG
profit_in_usd = 0.9 * profit_in_eur

# CORRECT
EUR_TO_USD_RATE = 0.9
profit_in_usd = EUR_TO_USD_RATE * profit_in_eur
```

Benennung von Integers (Ganzzahlen)

Die Benennung von Integers sollte nach PEP-8 [68] dem Schema des lower snake_case entsprechen, welcher besagt, dass alle Buchstaben kleingeschrieben und Abstände durch Underscores „_“ ersetzt werden. Wenn die Einheit des Integers nicht selbsterklärend ist, sollte

diese am Ende des Namens angehängt werden (siehe `retry_delay_in_ms`). Im Zweifel sollte die Einheit sicherheitshalber angegeben werden.

```
# WRONG
failures = 10
retry_delay = 500
distance = 33
price = 33
```

```
# CORRECT
failure_count = 10
retry_delay_in_ms = 500
distance_in_km = 33
price_in_eur = 33
```

Benennung von Floats (Gleitkommazahlen)

Ident zu den bereits beschriebenen Integers, sollte die Benennung von Floats nach PEP-8 [68] ebenfalls dem Schema des lower snake_case folgen. Darüber hinaus sollte die Einheit am Ende des Namens angeführt werden, wenn diese nicht selbsterklärend ist.

```
# WRONG
snowfall = 5.2
distance = 33.0
distance = 33.0
angle = 45.0
failure = 66.0
failure = 0.5
```

```
# CORRECT
snowfall_amount_in_mm = 5.2
distance_in_km = 33.0
angle_in_degrees = 45.0 # values from 0 to 360
failure_percent = 66.5 # values from 0 to 100
failure_ratio = 0.5 # values from 0 to 1
```

Hinweis: Seit dem Jahre 2000 nutzen die meisten Rechner eine vereinheitlichte binäre Gleitkommaarithmetik nach IEEE 754 [112]. Da einige Gleitkommazahlen aber nicht genau in einem binären Format dargestellt werden können, ergeben sich Rundungsfehler. Folgendes Beispiel soll dieses Problem verdeutlichen:

```
print(0.2 + 0.1)
```

Obwohl als Ergebnis `0.3` erwartet wird, gibt die Konsole `0.3000000000000004` aus. Aus diesem Grund ist es empfehlenswert sensible Variablen, wie beispielweise einen Preis, in Cent als Integer darzustellen. Eine detaillierte Erklärung der Probleme und Einschränkungen in Zusammenhang mit der Gleitkommaarithmetik gibt es von der Python Foundation [113].

Benennung von Booleans (booleschen Werten)

Die Benennung von Booleans sollte nach PEP-8 [68] ebenfalls anhand des Schemas des lowercase snake_case erfolgen.

Der Wert einer booleschen Variable kann entweder True oder False sein (None wird hier nicht berücksichtigt). Deswegen sollte die Benennung einer Variable so vorgenommen werden, dass sie eine konkrete Beantwortung einer binären Frage erlaubt - entweder Ja (True) oder Nein (False).

Insbesondere beim Lesen von if-Statements ist die gewählte Reihenfolge des Verbs und des Adjektivs bzw. Nomens fundamental. Das Verb sollte grundsätzlich in seiner aktiven Form geschrieben werden und vor dem Adjektiv oder Nomen stehen. Häufig verwendete Beispiele sehen laut Silen [104] wie folgt aus:

- **is_<something>**
- **has_<something>**
- **did_<something>**
- **should_<something>**
- **will_<something>**

Um einen noch besseren Lesefluss in Kombination mit dem if-Statement zu ermöglichen, kann die boolesche Variable umstrukturiert werden. Anstelle der bisherigen Form *Verb_Nomen_Adjektiv*, wird das Nomen mit dem Verb vertauscht, wodurch sich die Form *Nomen_Verb_Adjektiv* ergibt.

```
# WRONG
if is_pool_full:
    # ...
```

```
# CORRECT
if pool_is_full:
    # ...
```

Um eine sinnvolle Interpretation gewährleisten zu können, sollte folgende Form strengstens vermieden werden:

- **<passives-verb>_something**

```
# WRONG
if inserted_table:
    # ...
```

```
# CORRECT
if table_was_inserted:
    # ...
```

Darüber hinaus kann es der Fall sein, dass eine negierte boolesche Variable eingesetzt wird. Während dies grundsätzlich kein Problem darstellt, sollte die Variable dennoch aus dem if-Statement herausgezogen und dementsprechend unbenannt werden.

```
# WRONG
machine_was_started = machine.start()
if not machine_was_started:
    # ...
```

```
# CORRECT
machine_was_not_started = not machine.start()
if machine_was_not_started:
    # ...
```

Werden mehrere binäre Abfragen im Rahmen eines if-Statements auf einer Programmzeile durchgeführt, so sollten diese aus Gründen der besseren Lesbarkeit als boolesche Variablen extrahiert bzw. herausgezogen werden.

```
# WRONG
if person.age > 18 and person.has_license:
    is_allowed_to_drive = True
```

```
# CORRECT
is_adult = person.age > 18
has_license = person.has_license
is_allowed_to_drive = is_adult and has_license
```

Benennung von Strings

Die Benennung von Strings sollte nach PEP-8 [68] ebenfalls dem Schema des lowercase snake_case entsprechen. Da allgemeine Benennungsregeln bzw. Konstanten bereits erläutert wurden und Type Hints bzw. f-Strings in einem anderen Kapitel folgen, wird dieser Abschnitt bewusst kurz gehalten.

```
# WRONG
UserName = "Amadeus"

# CORRECT
user_name = "Amadeus"
```

Darüber hinaus sollten Anführungszeichen “ einheitlich eingesetzt werden:

```
# WRONG
user_names = ["Amadeus", 'Ludwig']

# CORRECT
user_names = ["Amadeus", "Ludwig"]
```

Benennung von Lists und Sets

Die Benennung von Listen und Sets sollte nach PEP-8 [68] ebenfalls anhand des Schemas des lowercase snake_case erfolgen.

Wie bereits in den allgemeinen Benennungsregeln beschrieben, sollte der Datentyp dem Variablenamen grundsätzlich nicht angehängt werden. Insbesondere bei Listen oder Sets wird instinktiv oft „*_list*“ oder „*_set*“ angehängt. Dieses Verhalten lässt sich aber durch den Einsatz des Plurals effektiv vermeiden.

```
# WRONG
customer_list = [...]
student_id_set = {"001", "002", "003"}
```

```
# CORRECT
customers = [...]
student_ids = {"001", "002", "003"}
```

Eine Ausnahme gilt, wie bereits erwähnt, wenn eine Variable in mehreren Datentypen verwendet wird. Ein klassisches Beispiel ist das Erstellen einer Liste aus einem String und einer anschließenden Entfernung von Duplikaten bzw. Sortierung der Elemente.

```
# CORRECT
fruits_string = "banana apple cherry banana"
fruits_list = fruits_string.split(" ")
fruits_set = set(fruits_list)
```

Benennung von Dictionaries

Die Benennung von Dictionaries sollte nach PEP-8 [68] ebenfalls anhand des Schemas des lowercase snake_case erfolgen.

Des Weiteren sollte anhand der Namensgebung auf einen Blick erkannt werden, worum es sich beim Key und dem dazugehörigen Value handelt. Je nach Anwendungsfall kann das Value dem Singular- oder der Pluralform folgen. Das Muster sollte deshalb folgendermaßen definiert werden:

- *key_to_value*

```
# WRONG
country_currencies = {
    "AT": "EUR",
    ...
}
```

```
# CORRECT
country_to_currency = {
    "AT": "EUR",
    ...
}
```

- `key_to_values`

```
# WRONG
country_currencies = {
    "AT": ["EUR", "USD"],
    ...
}
```

```
# CORRECT
country_to_currencies = {
    "AT": ["EUR", "USD"],
    ...
}
```

In obigem Beispiel würde sich die Pluralform dann anbieten, wenn ein Land mehrere Währungen nutzen würde.

Benennung von Tuples

Die Benennung von Tuples sollte nach PEP-8 [68] ebenfalls anhand des Schemas des lowercase snake_case erfolgen.

Bei Tuples handelt es sich um einen unveränderlichen (immutable) Datentyp, welcher sich dazu eignet, mehrere zusammengehörige Werte in einer bestimmten Reihenfolge zu speichern. Es handelt sich dabei aber nicht zwangsläufig um Konstanten, da die Variable jederzeit auf einen neuen Wert gesetzt werden kann.

```
# CORRECT
height_and_width_in_mm = (100, 200)
coordinates = (48.2394, 16.3778)
```

Ist es der Fall, dass bei der Extrahierung der Werte aus einem Tuple, ein Wert nicht benötigt wird, dann sollte dieser mithilfe einer Throwaway-Variable „_“ symbolisiert werden.

```
locations = [
    ("Vienna", 48.2394, 16.3778),
]

for city, _, _ in locations:
    print(city)
```

Benennung von Klassen (bzw. Attributen und Methoden)

Ungleich zu den bisherigen Variablen soll die Benennung von Klassen nach PEP-8 [68] mithilfe des Schemas des PascalCase erfolgen. Dieses Schema besagt, dass jedes Wort im Namen mit einem Großbuchstaben beginnt und keine Trennzeichen, wie Unterstriche, verwendet werden dürfen.

Darüber hinaus bedarf es bei der Diskussion von Klassen, auch der Berücksichtigung von Klassenattributen. Grundsätzlich gibt es in Python keine technische Notwendigkeit zur Differenzierung zwischen privaten und öffentlichen Attributen, aber es ist essenziell, um zu erkennen, welche Attribute intern verwendet werden und welche von außen zugänglich sind. Diese trägt letztlich zu einer besseren Lesbarkeit bei.

- Das erste Argument der Methode einer Klasse muss `self` benannt werden.
- Public Attribute und Methoden dürfen nicht mit einem Unterstrich `_` beginnen.
- Private Attribute und Methoden müssen mit einem Unterstrich `_` beginnen.

```
# CORRECT
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self._balance = balance

    def get_balance(self):
        return self._balance

    def deposit(self, amount):
        if self._is_valid_amount(amount):
            self._balance += amount

    def _is_valid_amount(self, amount):
        return amount > 0
```

- Das erste Argument einer Klassenmethode muss `cls` benannt werden.

```
# CORRECT
class Machine:
    count = 0

    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        Machine.count += 1

    @classmethod
    def get_count(cls):
        return cls.count
```

- Die Klassenattribute sollten den Klassennamen nicht wiederholen.

```
# WRONG
class Order:
    def __init__(self, order_id, order_state):
        self.order_id = order_id
        self.order_state = order_state
```

```
# CORRECT
class Order:
    def __init__(self, id_, state):
        self.id_ = id_
        self.state = state
```

Achtung: `@classmethod` sollte nicht mit `@staticmethod` vertauscht werden. Während die `@classmethod` Zugriff auf die Klasse hat und demnach Klassenattribute und Methoden aufrufen oder modifizieren kann, gilt dies für die `@staticmethod` nicht. Typische Anwendungsbeispiele sind:

- `@classmethod` → Wenn die Methode Informationen über die Klasse benötigt.
- `@staticmethod` → Hilfsfunktionen, die keine Klassen- oder Instanzdaten benötigen.

Benennung von Objects (Objekten)

Die Benennung von Objekten, also den Instanzen von Klassen, sollte nach PEP-8 [68] ungleich zu Klassen anhand des Schemas des lowercase snake_case erfolgen.

Wenn möglich, sollte der Name des Objektes den Namen der zugrundeliegenden Klasse widerspiegeln. Darüber hinaus kann er durch Adjektive oder beschreibende Begriffe konkretisiert werden, um die spezifische Funktion oder Rolle des Objekts eindeutiger hervorzuheben.

```
# CORRECT
class Task:
    def __init__(self, name):
        self.name = name

build_task = Task("Build")
```

In manchen Fällen, insbesondere bei allgemeinen Konzepten, ist die explizite Verwendung des Klassennamens innerhalb des Objektnamens nicht möglich oder sinnvoll. Hier sollte die Rolle des Objektes anhand seines Namens und des gegebenen Kontexts klar identifizierbar sein.

```
# CORRECT
class Fruit:
    def __init__(self, name, color):
        self.name = name
        self.color = color

apple = Fruit("Apple", "Red")
banana = Fruit("Banana", "Yellow")
```

3.1.5 Type Hints

Bei Python handelt es sich um eine stark, dynamisch typisierte Programmiersprache.

Stark typisiert heißt, dass keine automatischen Typumwandlungen durchgeführt werden und der Datentyp bei der Durchführung von Operationen berücksichtigt wird. So können beispielsweise Integers und Floats miteinander addiert werden, Integers und Strings aber nicht.

Dynamisch typisiert bedeutet, dass die Datentypen der verwendeten Variablen automatisch zur Runtime erkannt werden. Das impliziert, dass sich ein Python-Entwickler nicht mit der expliziten Deklaration des Datentyps von Variablen, Parametern oder Return Values beschäftigen muss. Diese Flexibilität zieht, im Vergleich zu statisch typisierten Programmiersprachen, jedoch einige Probleme mit sich:

- **Typkonflikte**

Aufgrund von Inkonsistenzen in der Typisierung können Fehler auftreten, welche schwer nachvollziehbar sind. Dies kann beispielsweise durch die falsche Reihenfolge oder den falschen Datentyp von Funktionsargumenten geschehen.

- **Falsche Erwartungen**

Durch die fehlende Deklarierung von Datentypen, kann das Verhalten bzw. das Return-Value einer Funktion missverstanden werden.

- **Unübersichtlicher Code**

Die Variablen sind meist weniger aussagekräftig, wodurch die Lesbarkeit vermindert wird.

Aus diesem Grund wurde mit Python 3.5 die Möglichkeit der Definition von Type-Hints mit der PEP-484 [114] eingeführt:

```
variable: type = value

def function_name(parameter_1: parameter_1_type) -> return_type:
    ...
```

Bei Type-Hints sei jedoch angemerkt, dass keine tatsächliche Überprüfung der Datentypen zur Runtime geschieht. Stattdessen kann auf freiwilliger Basis ein externer Type-Checker, wie Mypy [115], verwendet werden, welcher wie ein sehr leistungsfähiger Linter funktioniert.

Demnach bietet die Deklaration von Type-Hints in Kombination mit der Nutzung eines Static-Type-Checkers eine optionale Möglichkeit potenzielle Fehler in einem Python-Code frühzeitig zu erkennen. Desto mehr Variablen, Parameter oder Return Values mit Type-Hints versehen werden, desto mehr Information stehen dem Type-Checker zur Verfügung, um präzise Analysen durchzuführen.

Variablen mit den dazugehörigen Type Hints versehen

Bei Type Hints muss zuallererst eine fundamentale Unterscheidung getroffen werden. Obwohl es stets empfehlenswert ist, immer die aktuellen Praktiken zu verwenden, kann es in praktischen Anwendungen, insbesondere bei der Kollaboration mit anderen Entwicklern, der Fall sein, dass nicht mit derselben Python-Version gearbeitet wird. Darüber hinaus ist es beispielweise bei der Wartung von älteren Projekten teilweise vielleicht gar nicht möglich ein Update auf eine neuere Python Version aus zeittechnischen, personellen beziehungsweise finanziellen Gründen durchzuführen. Des Weiteren kann es trotz gegebenen Mitteln problematisch sein auf eine aktuellere Python-Version upzugraden, da es Abhängigkeiten der importieren externen Module mit der Python-Version gibt. Entscheidet beispielweise ein Entwickler eines externen Moduls dazu seinen Programmcode upzudaten und diesen damit auf den aktuellen Stand der Technik zu bringen oder neue Features hinzuzufügen, kann es durchaus der Fall sein, dass dabei auf grundlegende Python-Funktionen zurückgegriffen werden, die erst ab einer bestimmten Version verfügbar waren. Würde deshalb nur Python oder eines der externen Module aktualisiert werden, würde es zu Kompatibilitätsproblemen kommen. Um solche Probleme zu vermeiden, wurden sogenannte Paketverwaltungssysteme entwickelt, wessen Aufgabe es ist, komplizierte Abhängigkeiten zu berücksichtigen. Hier sei jedoch am Rande angemerkt, dass die Nutzung von Paketverwaltungssystemen, wie Anaconda, zwar sehr beliebt sind, aber in der Produktionsumgebung aufgrund des Umfangs, der Größe und der Portabilität meist nicht eingesetzt werden.

Aus genannten Gründen wurde sich im Rahmen dieser Arbeit bewusst für eine Bearbeitung von Python < 3.10 (also von 3.5 bis 3.9) und Python 3.10+ (also von 3.10 bis 3.12) entschieden. Für beide Versionsgruppen gilt jedoch zumindest dieselbe Definition der klassischen Datentypen:

```
# WRONG
first_name = "Amadeus"
age = 66
success_rate = 0.95
machine_has_stopped = False
payload = b"Sensor_Data"
```

```
# CORRECT
first_name: str = "Amadeus"
age: int = 66
success_rate: float = 0.95
machine_has_stopped: bool = False
payload: bytes = b"Sensor_Data"
```

Seit Python 3.10 wurde die Verwendung von Union Types massiv vereinfacht. Statt einem Import von *Union* aus dem *typing*-Modul, kann direkt das eingebaute Zeichen „|“ verwendet werden. Grundsätzlich wurden die eingebauten Type Hints *dict*, *list*, *set*, *tuple* und *list* bereits mit Python 3.9 eingeführt, aber um Klarheit zu bewahren, wurde sich bewusst für oben

genannte Trennung entschieden. Vor der Python Version 3.10 werden Type Hints aus dem *typing*-Modul importiert. Ab der Python Version 3.10 werden die eingebauten Type Hints verwendet:

Python 3.10+:

```
last_name_to_age: dict[str, int] = {"Mozart": 66, "Beethoven": 99}
grades: list[int] = [1, 4, 2, 2, 3]
prime_numbers: set[int] = {2, 3, 5, 7}
screen_resolution: tuple[int, int] = (1920, 1080)
movie_ratings: list[float | str] = [1.5, "Great movie!", "N/A", 3.5]
username: str | None = get_username(user_id) if user_is_logged_in(user_id)
else None
```

Python <3.10 (3.5 bis 3.9):

```
from typing import Dict, List, Optional, Set, Tuple, Union

last_name_to_age: Dict[str, int] = {"Mozart": 66, "Beethoven": 99}
grades: List[int] = [1, 4, 2, 2, 3]
prime_numbers: Set[int] = {2, 3, 5, 7}
screen_resolution: Tuple[int, int] = (1920, 1080)
movie_ratings: list[Union[float, str]] = [1.5, "Great movie!", "N/A", 3.5]
username: Optional[str] = get_username(user_id) if user_is_logged_in(user_id)
else None
```

Funktionsparameter und Rückgabewerte mit den dazugehörigen Type Hints versehen

Das Versehen einer Funktion oder Methode mit Type Hints erfolgt analog zu den bereits beschriebenen Variablen mit dem Unterschied, dass bei Funktionen und Methoden auch ein Rückgabewert definiert werden muss. Dies geschieht mit der Einfuhr eines Pfeils „->“ gefolgt vom Datentyp des Rückgabewerts.

```
# Python 3.10+
def find_maximum(values: list[int]) -> int | None:
    if values:
        return max(values)
    else:
        return None
```

```
# Python <3.10
from typing import List, Optional

def find_maximum(values: List[int]) -> Optional[int]:
    if values:
        return max(values)
    else:
        return None
```

Ist eine Funktion oder Methode jedoch so gestaltet worden, dass sie lediglich eine Aktion durchführt und keinen Rückgabewert ausgibt, dann sollte dies auf einen Blick anhand von `None` ersichtlich sein.

```
def log_error(error_message: str) -> None:  
    print(f"[ERROR] {error_message}")
```

Hinweis: Globale Variablen dürfen nicht mit Type Hints versehen werden!

Das Ziel dieser Arbeit ist es, einen pragmatischen Leitfaden zu entwickeln, weshalb ganz bewusst nur die relevantesten Type Hints erläutert wurden. Bei Bedarf oder Interesse sollte das Python Enhancement Proposal 484 [114] oder die offizielle Mypy-Dokumentation [115] konsultiert werden. Abschließend sei jedoch angemerkt, dass es eine Vielzahl an weiteren Type Hints gibt, die im Rahmen dieser Arbeit nicht erwähnt werden (zum Beispiel `All`, `Iterable` oder `Sequence`). In den meisten Fällen werden diese Type Hints verwendet, wenn der Datentyp der Variable unbekannt ist und deshalb flexibel sein soll. Es mag besondere Anwendungsfälle geben, aber im Allgemeinen ist die Eingrenzung des Datentyps mithilfe von Union Types die bessere und vor allem nachvollziehbarere Lösung. Denn eine Nutzung von abstrakten bzw. allgemeinen Konstrukten geht mit einer sorgfältigen Prüfung des Datentyps einher, wodurch die Lesbarkeit des Programmcodes maßgeblich einschränkt wird.

3.1.6 Importing Modules

Während die flexible Nutzung von einer Bandbreite an Modulen ein großer Pluspunkt von Python ist, bedarf es einem grundlegenden Verständnis, wie ein korrekter Import zu erfolgen hat:

Imports am Anfang der Datei durchführen und niemals innerhalb einer Funktion

Um eine optimale Nachvollziehbarkeit gewährleisten zu können, sollten alle Module stets am Anfang einer Datei importiert werden. Dadurch wird versichert, dass Module nicht unabsichtlich mehrmals importiert werden und die Performance nicht durch einen dynamischen Import innerhalb einer Funktion negativ beeinflusst wird.

```
# WRONG
import module

def function_name() -> None:
    import another_module
    import module

    ...
```

```
# CORRECT
import another_module
import module

def function_name() -> None:
    ...
```

Imports auf separaten Zeilen durchführen und alphabetisch sortieren (oben nach unten)

Um die importierten Module übersichtlich darzustellen, sollten diese einerseits auf separaten Zeilen durchgeführt und alphabetisch sortiert werden. Damit wird es anderen Entwicklern ermöglicht, insbesondere bei Python-Skripten mit sehr vielen Imports, einfacher nachzuvollziehen, wo sie welches Modul finden. Dies gestaltet letztlich den Debugging-Prozess effizienter.

```
# WRONG
import module, another_module
```

```
# CORRECT
import another_module
import module
```

Die Anzahl an importieren Komponenten eines Moduls auf drei begrenzen

Durch den expliziten Import von mehreren Funktionen oder Klassen eines Moduls, kann die Lesbarkeit maßgeblich verschlechtert werden. Im Rahmen dieser Arbeit wurde deshalb die maximale Anzahl an expliziten Imports von Komponenten aus einem Modul mit drei begrenzt.

Es gibt hier keinen vorgeschriebenen Maximalwert, da die Benennung und damit auch die Zeichenlänge variiert. Jedoch lassen sich generelle Aussagen auf Basis der Speicherkapazität des menschlichen Gedächtnisses treffen. Miller [106] stellte im Jahre 1956 fest, dass ein Mensch in etwa 7 ± 2 Dinge im Kurzzeitgedächtnis abspeichern kann. Cowan [107] hat im Jahre 2001 auf diesen Erkenntnissen aufgebaut und herausgefunden, dass es sich dabei nur um einen groben Richtwert handelt und deswegen den tatsächlichen Wert auf 4 ± 1 begrenzt. Der untere Grenzwert wird deshalb als Maximalwert an expliziten Importen gewählt.

```
# WRONG
from module import component_1, component_2, component_3, component_4

# CORRECT
import module
```

Umgekehrt gilt Analoges: Das heißt werden maximal drei Komponenten verwendet, so empfiehlt es sich, diese auch explizit zu importieren.

```
# WRONG
import module

module.component_1()
module.component_2()
module.component_3()

# CORRECT
from module import component_1, component_2, component_3
```

Die einzige Ausnahme sind Imports aus dem Modul *typing*, welche zur optionalen Definition von Type Hints benötigt werden. Hier wird keine maximale Anzahl an expliziten Imports berücksichtigt.

```
from typing import Dict, List, Optional, Set, Tuple, Union
```

Importierte Komponenten alphabetisch sortieren (links nach rechts)

Um die importierten Komponenten übersichtlich darzustellen, sollten diese ebenfalls alphabetisch sortiert werden. Auch wenn diese Konvention lediglich einen marginalen Beitrag zur verbesserten Lesbarkeit leistet, gestaltet sie den Debugging-Prozess effizienter.

```
# WRONG
from module_1 import send_notification_email, EventLogger, EmailValidator

# CORRECT
from module_1 import EmailValidator, EventLogger, send_notification_email
```

Lange Modulnamen durch sinnvollen Alias ersetzen

Um die Verwendung von langen Modulnamen und eine damit verbundene Unübersichtlichkeit bzw. erhöhte Schreibarbeit zu vermeiden, sollten sinnvolle und aussagekräftige Aliase verwendet werden.

```
# WRONG
import very_very_long_module_name
```

```
# CORRECT
import very_very_long_module_name as alias
```

Oftmals ist es auch sinnvoll etablierte Aliase von häufig verwendeten Modulen zu verwenden.

```
# CORRECT
import matplotlib.pyplot as plt
import pandas as pd
```

Teilweise sind Aliase auch zwingend notwendig, wenn verschiedene Module dieselbe Funktions- oder Klassenbenennung besitzen.

```
# WRONG
from module_1 import component
from module_2 import component
```

```
# CORRECT
from module_1 import component as component_1
from module_2 import component as component_2
```

Vermeiden von Wildcard-Imports

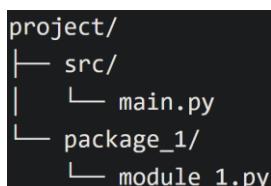
Der Einsatz von Wildcard-Import sollte strengstens vermieden werden, da im Programmcode nicht ersichtlich wird, aus welchem Modul die verwendeten Funktionen- bzw. Klassen entstammen. Da der Ursprung der Komponenten unbekannt ist, wird der Debugging-Prozess massiv erschwert. Des Weiteren kann ein Wildcard-Import einen Namespace-Konflikt verursachen, da alle Komponenten ohne dazugehöriges Modul importiert werden.

```
# WRONG
from module import *
```

```
# CORRECT
from module import component
```

Vermeiden von Relative-Imports

Relative Imports sollten stets vermieden und durch absolute Imports ersetzt werden. Damit der Unterschied deutlicher und nachvollziehbar wird, bedarf es hier der zusätzlichen Angabe eines Verzeichnisses.



Bei relativen Imports symbolisiert ein Punkt, dass auf den aktuellen Ordner verwiesen wird, zwei Punkte, dass auf den übergeordneten Ordner verwiesen wird, und immer so weiter. Wird die Verzeichnisstruktur geändert, muss jeder Punkt in den Import-Statements überprüft und gegebenenfalls angepasst werden.

```
# WRONG
from ..package_1.module_1 import component_1
```

Bei absoluten Imports jedoch ist die Verzeichnisstruktur flexibel, solange Python das Wurzelverzeichnis erkennt. Der einzige Nachteil kann ein langes Import-Statement sein, wenn die Ordnerstruktur sehr verschachtelt ist, wobei diese bei Bedarf sowieso vereinfacht und umstrukturiert werden sollte.

```
# CORRECT
from package_1.module_1 import component_1
```

Einteilung der importierten Module in Standard, Third Party und Local Imports

Grundsätzlich wird zwischen drei Arten von Imports differenziert:

1. Standard library imports
2. Third party imports
3. Local imports

Um auf einen Blick klar unterschieden zu können, welches Module welcher Kategorie zugeordnet wird, sollten die Module in eben diese Gruppen gegliedert werden. Hierbei empfiehlt es sich, je nach Entwicklungserfahrung, bewusst auf den Einsatz von Block-Kommentaren zu setzen. Innerhalb jeder Gruppe gelten sämtliche bereits genannte Regeln, mit dem Zusatz, dass das Ende einer Gruppe mit einer Blank Line signalisiert wird.

```
# WRONG
import requests
from local_module import *
import time
from flask import Flask
import os
```

```
# CORRECT
# Standard library imports
import time
import os

# Third party imports
from flask import Flask
import requests

# Local imports
from local_module import local_component
```

Entfernen von ungenutzten Modulen/Komponenten

Jene Module bzw. Komponenten, die im Programmcode nicht verwendet werden, sollten konsequent entfernt werden. Dadurch profitiert der Code nicht nur an verbesserter Lesbarkeit, sondern ermöglicht auch einen geringeren Speicherverbrauch und eine kürzere Startzeit.

```
# WRONG
from module_1 import function_1
from module_2 import Class_2, function_2

function_2()
```

```
# CORRECT
from module_2 import function_2

function_2()
```

3.1.7 Docstrings

Ein Docstring wird nach PEP-257 [116] als eine Zeichenkette definiert, welche die erste Anweisung in einem Modul, einer Klasse, einer Methode oder einer Funktion ist. Der Docstring ist ein spezielles Attribut „`__doc__`“ des beschriebenen Objektes. Insbesondere bei der Einarbeitung in fremden Code und einem damit verbundenen Debugging ist diese Konvention eine hilfreiche Methodik. In modernen Entwicklungsumgebungen, sogenannten Integrated Development Environments (IDEs), ermöglicht das Hovern mit der Maus über einer Klasse, einer Methode oder einer Funktion das Anzeigen des Docstrings.

Es gibt zwei Arten von Docstrings: Einerseits one-line und andererseits multi-line Docstrings. One-line Docstrings dürfen nach PEP-257 [116] nur bei sehr offensichtlichen bzw. selbsterklärenden Fällen eingesetzt werden. Obwohl one-line Docstrings auch mit einfachen Anführungszeichen geschrieben werden könnten, ist für eine spätere Erweiterung die Verwendung von dreifachen Anführungszeichen grundvoraussetzend. Da die Definition von Selbsterklärend im Auge des Betrachters liegt und dadurch einen unerwünschten Interpretationsrahmen ermöglicht, werden im Leitfaden aus Gründen der Einheitlichkeit nur multi-line Docstrings berücksichtigt.

Allgemeine Docstring-Regeln

Die PEP-257 [116] setzt folgende Maßnahmen voraus:

- Vor dem Docstring darf es keine Leerzeile geben.

```
# WRONG
def function_name(...) -> return_type:

    """
    Docstring goes here.
    """
    return return_value
```

```
# CORRECT
def function_name(...) -> return_type:
    """
    Docstring goes here.
    """
    return return_value
```

- Die Einrückung des Docstrings sollte mit dem dazugehörigen Code übereinstimmen.

```
# WRONG
def function_name(...) -> return_type:
    """Docstring goes here.
    """
    return return_value
```

```
# WRONG
def function_name(...) -> return_type:
    """
    Docstring goes here."""
    return return_value
```

```
# CORRECT
def function_name(...) -> return_type:
    """
    Docstring goes here.
    """
    return return_value
```

- Der Docstring muss einen Satz darstellen und daher mit einem Punkt „.“ enden. Zudem sollte er das Verhalten der Funktion/Methode als Befehl erklären (Do X & Return Y.) und keine passive Beschreibung darstellen (Returns X).

```
# WRONG
def add_numbers(a: int, b: int) -> int:
    """
    Returns sum of two numbers
    """
    return a + b
```

```
# CORRECT
def add_numbers(a: int, b: int) -> int:
    """
    Add two numbers and return their sum.
    """
    return a + b
```

- Der Docstring darf inhaltlich keine redundante Wiederholung der Funktion oder der Methode und dessen Parameter darstellen.

```
# WRONG
def add_numbers(a: int, b: int) -> int:
    """
        add_numbers(a, b) -> int
    """
    return a + b
```

```
# CORRECT
def add_numbers(a: int, b: int) -> int:
    """
        Add two numbers and return their sum.
    """
    return a + b
```

- Aus Gründen der Konsistenz sollte der Inhalt eines Docstrings zwischen "'''triple double quotes''' geschrieben werden. Ergibt sich die Notwendigkeit, dass Backslashes (\) im Docstring vorkommen müssen, so dürfen r'''raw triple double quotes''' verwendet werden.

```
# WRONG
def read_file(file_path: str) -> str:
    """
    ...
    Examples
    -----
    >>> read_file(r"C:\Users\Username\new_folder\file.txt")
    Hello there!
    """
    with open(file_path, 'r') as file:
        return file.read()
```

```
# CORRECT
def read_file(file_path: str) -> str:
    r"""
    ...
    Examples
    -----
    >>> read_file(r"C:\Users\Username\new_folder\file.txt")
    Hello there!
    """
    with open(file_path, 'r') as file:
        return file.read()
```

Docstrings für Funktionen & Methoden

- Nach dem Funktionen-/Methoden-Docstring darf es keine Leerzeile geben.

```
# WRONG
def function_name(...) -> return_type:
    """
    Docstring goes here.
    """

    return return_value

# CORRECT
def function_name(...) -> return_type:
    """
    Docstring goes here.
    """

    return return_value
```

- Bei der Dokumentation von Methoden darf `self` nicht als Parameter angeführt werden.

```
# WRONG
class Calculator:
    def add(self, a: int, b: int) -> int:
        """
        ...

        Parameters
        -----
        self : Calculator
            The instance of the class.
        a : int
            First number.
        b : int
            Second number.

        ...
        """
        return a + b
```

```
# CORRECT
class Calculator:
    def add(self, a: int, b: int) -> int:
        """
        ...

        Parameters
        -----
        a : int
            First number.
        b : int
            Second number.

        ...
        """
        return a + b
```

Die Panda Docstring-Richtlinie [117], welche auf der PEP 287: „reStructuredText Docstring Format“ [118] beruht, empfiehlt folgende Einteilung mit der Trennung durch Bindestriche "--":

- Short summary
- Extended summary (if function is not too generic)
- Parameters
- Returns or yields
- ~~See also~~
- ~~Notes~~
- Examples

Damit der Docstring prägnant bleibt und sich auf die wesentlichen Kategorien fokussiert, werden die optionalen Abschnitte „See Also“ und „Notes“ nicht berücksichtigt.

Werden die Clean-Code-Prinzipien der PEP-257 [116] mit der strukturierten Einteilung der Panda/Numpy Richtlinie [117] kombiniert und in eine Vorlage zusammengefasst, so ergibt sich eine Docstring-Struktur, die sowohl konsistent als auch einfach nachvollziehbar ist.

```
def function_name(param1: param1_type, param2: param2_type) -> return_type:  
    """  
        Brief description of the function as an imperative command ("Do X & return Y")  
  
        Detailed explanation of what the function does.  
        Why is it needed? Are Error raised? If so, when?  
        What steps or calculations does it perform?  
  
    Parameters  
    -----  
    param1: param1_type  
        Description of the first parameter.  
    param2: param2_type  
        Description of the second parameter.  
  
    Returns  
    -----  
    return_type  
        Description of what the function returns.  
  
    Raises (if applicable)  
    -----  
    <ErrorType>  
        Description of the exception raised if an error occurs.  
  
    Examples  
    -----  
    >>> function_name(param1_value, param2_value)  
    <expected_return_value_of_given_example>  
    """  
    return return_value
```

Damit die Anwendung der Funktionen/Methoden-Docstring-Vorlage demonstriert werden kann, wird folgendes Beispiel angeführt:

```
def calculate_total_with_tax(prices: list[int | float], tax_rate: float) -> float:
    """
    Sums up given prices, applies the given tax rate, and returns the total price.

    This function sums up the provided price list and applies the given tax rate.
    It raises a ValueError if the tax rate is negative or if any price is
    either non-numeric or negative.

    Parameters
    -----
    prices: list[int | float]
        A list of item prices.
    tax_rate: float
        The tax rate to apply.

    Returns
    -----
    float
        The total price after applying the tax.

    Raises
    -----
    ValueError
        If the tax rate is negative, or
        if any price is non-numeric or negative.

    Examples
    -----
    >>> calculate_total_with_tax([100.2, 50, 25.5], 0.1)
    193.27
    """
    if tax_rate < 0:
        raise ValueError(f"Invalid tax rate: {tax_rate}. Must be a non-negative value.")

    prices_are_non_numeric = not all(
        isinstance(price, (int, float)) for price in prices
    )
    if prices_are_non_numeric:
        raise ValueError("All prices must be of numeric (int/float)")

    prices_are_negative = any(price < 0 for price in prices)
    if prices_are_negative:
        raise ValueError("All prices must be non-negative.")

    total: float = sum(prices)
    total_with_tax = total * (1 + tax_rate)
    return total_with_tax
```

Docstrings für Klassen

Die Verfassung von Docstrings erfolgt nach der Numpy Docstring-Richtlinie [119] ähnlich zu denen von Funktionen bzw. Methoden, mit dem Unterschied, dass Klassen eine andere Aufteilung benötigen. So werden die Kategorien, „Return or yields“, „Parameters“ und „Examples“ entfernt, dafür aber die Abschnitte „Attributes“ und „Methods“ hinzugefügt:

- Short summary
- Extended summary (if class is not too generic)
- Attributes
- Methods

Wichtig ist es, alle public Klassenattribute im Docstring der Klasse zu dokumentieren, insbesondere diejenigen, die nicht explizit in der `__init__`-Methode als Parameter verwendet werden. Das heißt, dass intern initialisierte Variablen, die beispielsweise als Standardwerte definiert werden, auch zu berücksichtigen sind.

```
# WRONG
class SomeClass:
    """
    ...
    Attributes
    -----
    attribute1: attribute1_type
        Description of the first attribute (initialized via the constructor).

    ...
    """

    def __init__(self, attribute1: attribute1_type) -> None:
        self.attribute1: attribute1_type = attribute1
        self.attribute2: attribute1_type = False
```

```
# CORRECT
class SomeClass:
    """
    ...
    Attributes
    -----
    attribute1: attribute1_type
        Description of the first attribute (initialized via the constructor).
    attribute2: attribute2_type
        Description of the second attribute (initialized internally).

    ...
    """

    def __init__(self, attribute1: attribute1_type) -> None:
        self.attribute1: attribute1_type = attribute1
        self.attribute2: attribute1_type = False
```

Die Kategorie “Methods” im Klassen-Docstring sollte weder `self` als Parameter noch private Methoden enthalten.

```
# WRONG
class SomeClass:
    """
    ...
    Methods
    -----
    method1(self, param1)
        Brief description of what method1 does.
    _private_method()
        This is a private method and should not be listed.

    ...
    """

    def __init__(self, attribute1: attribute1_type) -> None:
        self.attribute1: attribute1_type = attribute1

    def _private_method(self) -> None:
        pass
```

```
# CORRECT
class SomeClass:
    """
    ...
    Methods
    -----
    method1(param1)
        Brief description of what method1 does.

    ...
    """

    def __init__(self, attribute1: attribute1_type) -> None:
        self.attribute1: attribute1_type = attribute1

    def _private_method(self) -> None:
        pass
```

Nach dem Klassen-Docstring muss es eine Leerzeile geben.

```
# WRONG
class SomeClass:
    """
    ...
    ...
    def __init__(self, attribute1: attribute1_type) -> None:
        self.attribute1: attribute1_type = attribute1
```

```
# CORRECT
class SomeClass:
    """
    ...
    ...
    def __init__(self, attribute1: attribute1_type) -> None:
        self.attribute1: attribute1_type = attribute1
```

Somit ergibt sich folgende Struktur für das Verfassen von Docstrings für Klassen:

```
class SomeClass:
    """
    Brief description of the class.

    Detailed explanation of what the class does.
    Why is it needed? Are Error raised? If so, when?
    What steps or calculations does it perform?

    Attributes
    -----
    attribute1: attribute1_type
        Description of the first attribute.
    attribute2: attribute2_type
        Description of the second attribute.

    Methods
    -----
    method1(param1)
        Description of what method1 does.
    """

    def __init__(self, attribute1: attribute1_type) -> None:
        self.attribute1: attribute1_type = attribute1
        self.attribute2: attribute1_type = False

    def method1(self, param1: param1_type) -> return_type:
        ...

    def _private_method(self) -> None:
        ...
```

Damit die Anwendung der Klassen-Docstring-Vorlage demonstriert werden kann, wird folgendes Beispiel angeführt:

```
class Machine:
    """
    A class used to represent a Machine.

    This class models a basic machine with functionality to start and stop.
    It keeps track of its running state.

    Attributes
    -----
    brand: str
        The brand of the machine.
    model: str
        The model of the machine.
    is_running: bool
        Indicates whether the machine is currently running.

    Methods
    -----
    start():
        Starts the machine, changing its state to running.
    stop():
        Stops the machine, changing its state to not running.
    """

    def __init__(self, brand: str, model: str) -> None:
        """
        Constructs all the necessary attributes for the Machine object.

        Parameters
        -----
        brand: str
            The brand of the machine.
        model: str
            The model of the machine.
        """
        self.brand: str = brand
        self.model: str = model
        self.is_running: bool = False

    def start(self) -> None:
        """
        Starts the machine.

        This method changes the state of the machine to running and outputs
        a message indicating that the machine has started.
        """
        self.is_running = True
        print(f"{self.brand} {self.model} started.")

    def stop(self) -> None:
        """
        Stops the machine.

        This method changes the state of the machine to not running and
        outputs a message indicating that the machine has stopped.
        """
        self.is_running = False
        print(f"{self.brand} {self.model} stopped.")
```

Docstrings für Module

Jedes Modul muss laut PEP 287: „reStructuredText Docstring Format“ [118] und Numpy Docstring-Richtlinie [120] ebenfalls einen Docstring im folgenden Format besitzen, wobei die einzelnen Kategorien mit Bindestrichen "----" voneinander getrennt werden:

- Module name
- Short summary
- Extended summary (if module is not too generic)
- ~~Routine listings (especially for large modules)~~
- ~~See also~~
- ~~Notes~~
- ~~References~~
- Examples

Damit der Docstring prägnant bleibt und sich auf die wesentlichen Kategorien fokussiert, werden die optionalen Abschnitte „See Also“, „Notes“ und „References“ nicht berücksichtigt. Da bereits Funktionen-/Methoden- bzw. Klassen-Docstrings definiert wurden und eine überflüssige Redundanz vermieden werden soll, wird auch der Abschnitt „Routine listings“ nicht mit einbezogen.

Nach dem Modul-Docstring muss es zwei Leerzeilen geben.

```
# WRONG
"""
Module name
=====
...
"""

class SomeClass:
    ...
```

```
# WRONG
"""
Module name
=====
...
"""

class SomeClass:
    ...
```

```
# CORRECT
"""
Module name
=====
...
"""

class SomeClass:
    ...
```

Für das Verfassen von Docstrings für Modulen ergibt sich deshalb folgende Struktur:

```
"""
Module name
=====

Brief description of the module.

Detailed explanation of the module (including its purpose and any other relevant
details)

Examples
-----
Provide example usage patterns.

>>> from module_name import some_function
>>> result = some_function(arguments)
>>> print(result)

>>> from module_name import SomeClass
>>> some_object = SomeClass(arguments)
>>> some_object.some_method()
"""

def some_function():
    ...

class SomeClass:
    ...
```

Folgendes Beispiel soll die Anwendung der Modul-Docstring-Vorlage exemplarisch demonstrieren:

```
"""
Machine utils
=====

A simple module for representing and managing machines.

This module provides a class to model a basic machine with the ability to start and stop.
It keeps track of the machine's running state and provides methods to control it.

Examples
-----
>>> from machine_utils import Machine
>>> machine_1 = Machine("Honda", "GX200")
>>> machine_1.start()
Honda GX200 started.

>>> machine_1.stop()
Honda GX200 stopped.
"""

class Machine:
    """
    ...

    def __init__(self, brand: str, model: str) -> None:
        ...
        self.brand: str = brand
        self.model: str = model
        self.is_running: bool = False

    def start(self) -> None:
        ...
        self.is_running = True
        print(f"{self.brand} {self.model} started.")

    def stop(self) -> None:
        ...
        self.is_running = False
        print(f"{self.brand} {self.model} stopped.")
    """
```

3.1.8 Formatting & spacing

Verwenden der korrekten Zeilenlänge

Da Black [121] ein *opinionated* Code-Formatter ist und keine Konfigurationsmöglichkeit der maximalen Zeilenlänge von 88 Zeichen erlaubt, wird autopep8 [122] verwendet. Da es aber auch bei diesem Linter keine Differenzierung zwischen regulären Codezeilen und Textblöcken gibt, wird die maximale Zeilenlänge mithilfe einer expliziten Regel nach PEP-8 [68] für reguläre Codezeilen auf 79 Zeichen und für lange Textblöcke, wie Docstrings oder Kommentare, auf 72 Zeichen begrenzt.

Abschließende Formatierung nach PEP-8 mittels autopep8

Hinweis: Diese Regel wird nicht mit einem LLM, sondern mit einem Linter angewendet!

Mithilfe von Lintern, wie autopep8 [122], ist es möglich ein Python-File in Einklang mit der PEP-8-Richtlinie zu bringen. Dies betrifft die Formatierung von Einrückungen, von Leerzeichen, von Leerzeilen, der Zeilenlänge und der einheitlichen Verwendung von doppelten Anführungszeichen ("") bei Strings. Obwohl bereits die meisten PEP-8 Formatierungsregeln indirekt durch die Definition von Beispielen im Leitfaden angewendet werden, wird ganz bewusst autopep8 [122] aufgrund seiner geringen Optimierungsdauer ergänzend eingesetzt.

3.2 Automatisierte Anwendung durch LLM-Prompting

Um dem Entwickler eine bequeme, zeitsparende und automatisierte Anwendung des Leitfadens auf ein gegebenes Python-File zu ermöglichen, ist die Einbindung eines Large-Language-Modells, wie einem von OpenAI, notwendig. Die Kommunikation mit dem LLM erfolgt dabei nicht über die jeweilige Benutzeroberfläche, sondern direkt über die bereitgestellte REST-API. Angemerkt sei hier jedoch, dass das Teilen von vertraulichen Dateien mit externen LLM-Anbietern ein möglicher Kritikpunkt sein könnte, weshalb Unternehmen eine interne Lösung in Betracht ziehen könnte. Obwohl einige Unternehmen einen verschlüsselten End Point besitzen, über welchen sämtliche Daten laufen, kann es dennoch dazu kommen, dass Nutzerdaten zu Verbesserungszwecken verwendet werden. Eine lokale Lösung, ohne Verwendung eines externen Anbieters, umfasst daher die Einbindung eines lokalen LLMs. Dies setzt jedoch entsprechende Rechenressourcen (z.B.: leistungsfähige Grafikkarten) sowie eine geeignete Infrastruktur voraus, was den Rahmen dieser Arbeit sprengen würde.

Im Rahmen dieser Arbeit werden die folgenden Flagship-Modelle von Anthropic, Google, OpenAI und Xai verglichen:

Tabelle 14: Vergleich der verwendeten LLMs

Model	Context Window	Pricing (Per Million Tokens)		Token Limit (Per Minute)	Request Limit (Per Minute)	Fine Tuning possible?
		Text Input	Text Output			
claude-3-7-sonnet-20250219 [123]	200,000	\$3.00	\$15.00	Input: 20,000 Output: 8,000	50	✗
gemini-2.0-pro-exp-02-05 [124]	2,000,000	\$1.25*	\$5.00*	1,000,000	5	✗
gpt-4o-2024-08-06 [125]	128,000	\$2.50	\$10.00	30,000	500	✓
grok-2-1212 [126]	131,072	\$2.00	\$10.00	N/A	240	✗

Wichtig: *Das es sich bei gemini-2.0-pro-exp-02-05 um ein experimentelles Modell handelt, welches derzeit kostenfrei zur Verfügung steht, wurden die Preise des Vorgängermodells gemini-1.5-pro-002 angenommen. Da in dieser Arbeit keine Iteration mehr als 128,000 Tokens benötigt, wurden Token-Preise gemäß der günstigeren Preisstufe ($\leq 128K$ Tokens) berechnet.

Das Context Window legt die maximal zulässige Gesamtzahl an Input- und Output-Tokens fest, die das Modell während eines Durchlaufs verarbeiten kann. Die Request Limits und Token Limits beziehen sich auf Tier-1-Mitgliedschaften von Chat Completions.

Der in dem Python-Skript (OptiPy) verwendete Leitfaden (siehe Anhang A: guideline.md) basiert auf dem in Kapitel 3.1 beschriebenen Clean-Code-Prinzipien. Es wurde sich für die

Verwendung eines Markdown-Files entschieden, da dieses dem LLM eine klar nachvollziehbare Struktur bietet und zeitgleich dem Menschen durch Markdown-Rendering eine flexible Modifizierbarkeit ermöglicht. In dieser Datei befindet sich eine prägnante Zusammenfassung der erläuterten Regeln, wobei der Leitfaden in die englische Sprache übersetzt wurde. Grundsätzlich gibt es zwei Möglichkeiten, den Leitfaden mithilfe des LLMs anzuwenden:

"All-at-Once"-Ansatz

- Alle Clean-Code-Kategorien werden in einem einzigen Durchgang angewendet.
- Die Reihenfolge der Regeln spielt daher keine entscheidende Rolle.
- Die konkrete Umsetzung inkl. Prompt ist dem Anhang B: optipy.py zu entnehmen.

"One-by-One"-Ansatz

- Jede Clean-Code-Kategorie wird in einem einzelnen Durchgang angewendet.
- Beginnend bei der ersten Kategorie wird dieser Prozess so lange wiederholt, bis alle Kategorien abgearbeitet wurden. Dies wird auch als **Prompt-Chaining** bezeichnet.
- Die Reihenfolge der Regeln kann daher eine entscheidende Rolle spielen.
- Die konkrete Umsetzung inkl. Prompt ist dem Anhang B: optipy.py zu entnehmen.

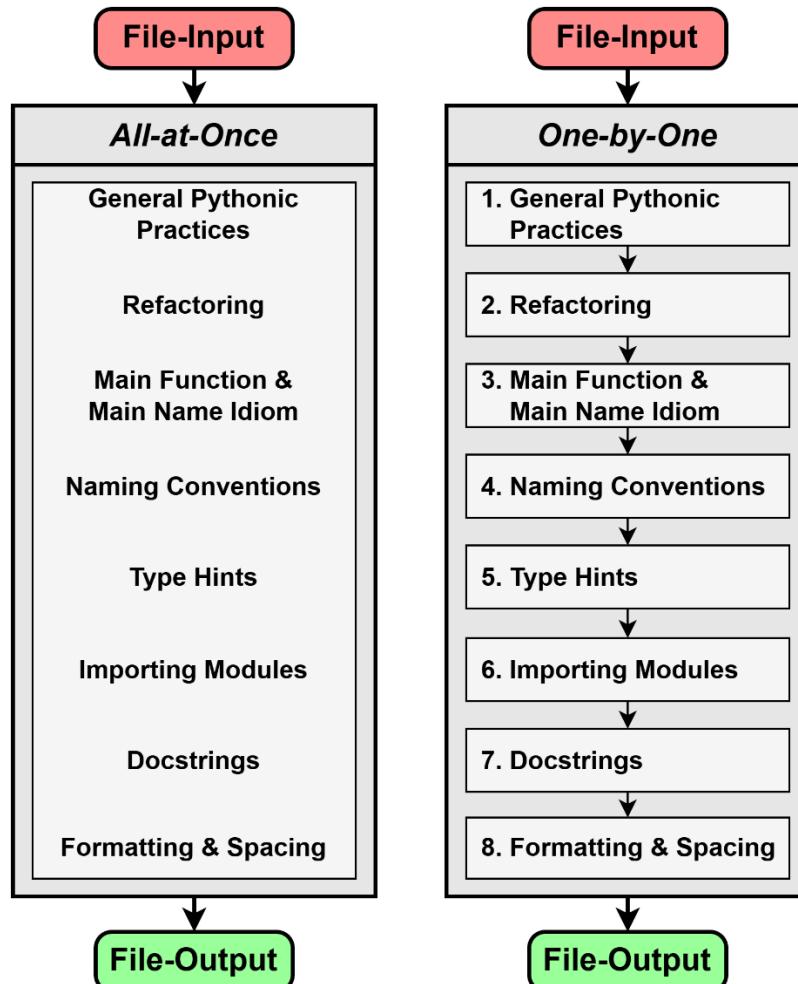


Abbildung 42: Ablaufdiagramm des "All-at-Once"- und des "One-by-One"-Ansatzes

Nach Gesprächen mit Industriepartnern stellt sich eine umfangreiche Bewertung des entwickelten Tools in der Praxis aufgrund der kurzfristigen Testphase von vier Monaten als Herausforderung dar. Um dennoch eine aussagekräftige Bewertung treffen zu können, wird auf den Einsatz von Fallstudien gesetzt. Dabei werden drei Anwendungsszenarien mit variierender Zeilenanzahl und für den jeweiligen Erfahrungsgrad typischen Fehler verwendet, um die Performance des Tools in verschiedenen Kontexten zu messen. Dabei ist insbesondere die Zeilenanzahl ein entscheidender Parameter, da sie sowohl die Belastbarkeit als auch die Skalierbarkeit des Tools in komplexen Produktionsumgebungen darstellt und auf mögliche Grenzen bzw. Einschränkungen hinweist.

Um aussagekräftige Rückschlüsse ziehen zu können, wird jede Case Study mehrmals durchgeführt. Die Ergebnisse werden anschließend mithilfe eines Vorher-/Nachher-Vergleichs evaluiert. Die konkrete Umsetzung der Evaluierung der verschiedenen Metriken ist dem Anhang C: metrics_analyzer.py zu entnehmen. Lediglich die Berechnung des Pylint-Scores benötigt eine nähere Erläuterung:

- `pylint --disable=W1203,R1705 example.py`

Dabei stehen die deaktivierten Regeln für:

- W1203 (logging-fstring-interpolation): Use %s formatting in logging functions
- R1705 (no-else-return): Unnecessary "elif" after "return", remove leading "el" from "elif"
Unnecessary "else" after "return", remove the "else" and de-indent the code inside it

Der Grund, warum die Regel W1203 nicht berücksichtigt wird, ist, dass die fstring-Interpolation in den meisten Anwendungen keinen signifikanten Einfluss auf die Effizienz hat, die Lesbarkeit jedoch maßgeblich erhöht. Ähnliches gilt für die Regel R1705: Obwohl ein else-statement nach einem return-Statement technisch irrelevant ist, erhöht es oft die Nachvollziehbarkeit der zugrundeliegenden Logik. Diese Regeln wurden somit bewusst deaktiviert, um eine vereinfachte und intuitivere Codegestaltung zu gewährleisten.

Danach soll mithilfe einer Nutzwertanalyse festgestellt werden, welches Large Language Model am besten für die Anwendung des entwickelten Leitfadens ist. Abschließend soll das gewählte Modell mithilfe eines Wirtschaftlichkeitsvergleich unter Berücksichtigung der Worst-Case, Neutral-Case und Best-Case Szenarien bewertet werden. Ein Wirtschaftlichkeitsvergleich ist essenziell, um aufzuzeigen, für welchen Sachverhalt das entwickelte Tool geeignet ist. Insbesondere die Frequenz, mit welcher die Entwickler:innen ihren Code auf Versionskontrollsysteme laden, spielt eine entscheidende Rolle. Hier soll hinterfragt werden, ob eine sinnvolle Nutzbarkeit des Tools im täglichen Gebrauch oder nur bei Meilensteinen gegeben ist.

Anmerkung: Die optimierten Python-Dateien der Fallstudien werden nicht im Anhang angeführt, um eine unnötige Erhöhung der Seitenanzahl zu vermeiden. Sämtliche Dateien sind jedoch im folgenden GitHub-Repository einsehbar: <https://github.com/phgas/optipy>

3.2.1 Fallstudie 1

Die erste Fallstudie beschäftigt sich mit einem Python-Code, der typische Fehler, sogenannte Code-Smells, eines unerfahrenen Entwicklers berücksichtigt:

Erfahrungsgrad	Anzahl an Regelverstößen	Code-Smells
Beginner	13	<ul style="list-style-type: none"> • Hohe Anzahl an redundanten Kommentaren • Unnötige Initialisierung der Variablen • Unklare Variablenbenennung • Nicht verwendeter Import • Inkonsistente Groß- und Kleinschreibung • Inkonsistente Abstände • Keine Verwendung von Funktionen • Keine Verwendung von Type Hints • Keine Verwendung von Docstrings • Inkonsistente Verwendung von Strings (" und ') • Keine Verwendung von f-Strings • Keine Verwendung des Main-Name-Idioms • Keine Verwendung von Logging

Programmcode vor der Optimierung (siehe Anhang E: case_study_1.py):

Tabelle 15: Auswertung der Fallstudie 1 vor der Optimierung

LOC	SLOC	Commented Lines	Average function length		Average CC	MI	Pylint Score
			with Docstrings	without Docstrings			
32	24	12	32.0	32.0	6.00 (B)	54.70 (A)	3.91

Programmcode nach der Optimierung mit dem „All-at-Once“-Ansatz:

Strategy	Reliability	Rule violations	Optimization time [s]		Token usage		PyLint Score	LOC	SLOC	Avg. Func. Len.		Commented Lines	Avg. CC	MI	Functional Correctness	Notable Findings
			Input	Output	Optimization costs	PyLint Score				with Docstrings	without Docstrings					
Iteration 1 ALL_AT_ONCE	✗	1	12.47	12810	828	\$ 0.051	9.71	126	43	20.50	8.75	0	2.50	75.74	✗	
Iteration 2 ALL_AT_ONCE	✗	1	13.78	12810	952	\$ 0.053	8.95	145	46	19.80	7.40	0	2.20	74.45	✗	
Iteration 3 ALL_AT_ONCE	✗	1	13.46	12810	873	\$ 0.052	9.73	137	45	18.20	7.40	0	2.20	75.15	✗	Logging not used properly
Iteration 4 ALL_AT_ONCE	✗	1	11.4	12810	753	\$ 0.050	8.92	127	45	16.20	7.80	0	2.20	74.66	✗	
Iteration 5 ALL_AT_ONCE	✗	1	13.62	12810	879	\$ 0.052	9.72	137	44	18.20	7.20	0	2.20	75.36	✗	
Average ALL_AT_ONCE	100%	1.0	12.95	12810	857	\$ 0.051	9.41	134.4	44.6	18.58	7.71	0	2.26	75.07	100%	
<hr/>																
Iteration 1 ALL_AT_ONCE	✗	0	9.25	12288	981	\$ 0.020	10.00	145	56	32.33	15.33	1	3.33	72.43	✗	
Iteration 2 ALL_AT_ONCE	✗	0	10.13	12288	1124	\$ 0.021	10.00	166	59	30.75	12.75	1	3.50	71.16	✗	
Iteration 3 ALL_AT_ONCE	✗	0	9.07	12288	1104	\$ 0.021	9.74	162	54	29.75	11.50	1	3.25	72.37	✗	(Long Input Messages)
Iteration 4 ALL_AT_ONCE	✗	0	10.76	12288	1141	\$ 0.021	10.00	177	55	31.25	11.50	1	3.25	72.62	✗	
Iteration 5 ALL_AT_ONCE	✗	0	10.46	12288	1108	\$ 0.021	10.00	175	57	29.25	12.00	1	3.25	72.28	✗	
Average ALL_AT_ONCE	100%	0.0	9.93	12288	1092	\$ 0.021	9.95	165.0	56.2	30.67	12.62	1	3.32	72.17	100%	
<hr/>																
Iteration 1 ALL_AT_ONCE	✗	3	17.93	10806	579	\$ 0.033	9.03	117	40	19.75	10.00	1	2.50	75.36	✗	
Iteration 2 ALL_AT_ONCE	✗	3	16.35	10806	580	\$ 0.033	9.03	117	40	19.75	10.00	1	2.50	75.36	✗	
Iteration 3 ALL_AT_ONCE	✗	2	15.30	10806	543	\$ 0.032	9.03	109	39	19.50	9.75	1	2.50	75.60	✗	
Iteration 4 ALL_AT_ONCE	✗	3	21.19	10806	579	\$ 0.033	9.03	117	40	19.75	10.00	1	2.50	75.36	✗	
Iteration 5 ALL_AT_ONCE	✗	3	21.04	10806	579	\$ 0.033	9.03	117	40	19.75	10.00	1	2.50	75.36	✗	
Average ALL_AT_ONCE	100%	2.8	16.26	10806	572	\$ 0.033	9.03	115.4	39.8	19.70	9.95	1	2.50	75.41	100%	
<hr/>																
Iteration 1 ALL_AT_ONCE	✗	4	13.49	10912	593	\$ 0.028	8.18	120	34	16.60	6.80	1	2.20	77.31	✗	
Iteration 2 ALL_AT_ONCE	✗	3	11.03	10912	547	\$ 0.027	8.67	110	31	19.50	8.00	0	2.50	78.15	✗	
Iteration 3 ALL_AT_ONCE	✗	3	10.86	10912	600	\$ 0.028	7.43	123	36	17.00	7.20	0	2.40	76.37	✗	
Iteration 4 ALL_AT_ONCE	✗	3	11.62	10912	571	\$ 0.028	8.79	121	35	16.80	7.00	0	2.20	76.66	✗	
Iteration 5 ALL_AT_ONCE	✗	3	12.53	10912	616	\$ 0.028	7.58	126	34	17.60	6.80	1	2.20	76.94	✗	
Average ALL_AT_ONCE	100%	3.2	11.91	10912	585	\$ 0.028	8.13	120.0	34	17.50	7.16	0.4	2.30	77.09	100%	

Programmcode nach der Optimierung mit dem „One-by-One“-Ansatz

Strategy	Reliability	Rule violations	Optimization time [s]	Token usage		Optimization costs	PyLint Score	LOC	SLOC	Avg. Func. Len.		Commented Lines	Avg. CC	MIL	Functional Correctness	Notable Findings
				Input	Output					with Docstrings	without Docstrings					
Iteration 1	ONE_BY_ONE	✗	0	74,62	17039	4533	\$ 0,119	10,00	138	42	31,00	11,00	1	2,67	75,25	✗
Iteration 2	ONE_BY_ONE	✗	0	59,76	17071	4598	\$ 0,120	10,00	151	43	35,33	11,33	1	2,67	75,03	✗
Iteration 3	ONE_BY_ONE	✗	0	60,14	17036	4526	\$ 0,119	10,00	138	43	31,00	11,33	1	2,67	75,03	✗
Iteration 4	ONE_BY_ONE	✗	0	63,15	17057	4568	\$ 0,120	10,00	142	43	32,33	11,33	1	2,67	75,03	✗
Iteration 5	ONE_BY_ONE	✗	0	61,98	17055	4565	\$ 0,120	10,00	147	43	34,00	11,33	1	2,67	75,03	✗
Average	ONE_BY_ONE	100%	0,0	63,93	17052	4558	\$ 0,120	10,00	143,2	42,8	32,73	11,26	1	2,67	75,07	100%
<hr/>																
Iteration 1	ONE_BY_ONE	✗	0	97,39	17605	7217	\$ 0,058	10,00	218	57	31,20	9,60	0	2,20	72,00	✗
Iteration 2	ONE_BY_ONE	✗	1	96,81	17355	6973	\$ 0,057	10,00	202	53	29,40	8,40	0	2,00	73,13	✗
Iteration 3	ONE_BY_ONE	✗	1	96,04	17360	6903	\$ 0,056	10,00	218	54	33,20	9,00	1	2,20	72,56	✗
Iteration 4	ONE_BY_ONE	✗	0	96,74	17844	7514	\$ 0,060	10,00	229	62	25,29	7,14	0	1,86	71,24	✗
Iteration 5	ONE_BY_ONE	✗	0	96,51	17319	6861	\$ 0,056	10,00	218	54	27,67	7,33	0	2,00	72,96	✗
Average	ONE_BY_ONE	100%	0,4	96,70	17497	7094	\$ 0,057	10,00	217,0	56	29,35	8,29	0,2	2,05	72,38	100%
<hr/>																
Iteration 1	ONE_BY_ONE	✗	0	101,84	14188	3289	\$ 0,068	10,00	105	41	21,33	10,00	1	3,00	75,43	✗
Iteration 2	ONE_BY_ONE	✗	0	115,64	14192	3193	\$ 0,067	10,00	102	43	15,75	8,00	1	2,50	75,05	✗
Iteration 3	ONE_BY_ONE	✗	0	113,53	14180	3296	\$ 0,068	10,00	101	41	21,00	10,00	1	3,00	75,43	✗
Iteration 4	ONE_BY_ONE	✗	0	134,85	14198	3525	\$ 0,071	10,00	110	41	22,67	11,00	1	3,00	75,43	✗
Iteration 5	ONE_BY_ONE	✗	0	79,77	14054	3241	\$ 0,068	10,00	101	43	15,50	8,00	1	2,50	75,05	✗
Average	ONE_BY_ONE	100%	0,0	109,13	14162	33088	\$ 0,068	10,00	103,8	41,8	19,25	9,40	1	2,80	75,28	100%
<hr/>																
Iteration 1	ONE_BY_ONE	✗	2	115,15	15801	6166	\$ 0,093	8,00	213	50	27,17	6,50	1	2,00	73,26	✗
Iteration 2	ONE_BY_ONE	✗	2	115,70	15743	5970	\$ 0,091	8,54	236	49	22,62	4,75	1	1,75	73,91	✗
Iteration 3	ONE_BY_ONE	✗	2	131,50	15996	6483	\$ 0,097	8,05	268	49	26,62	4,75	1	1,75	73,91	✗
Iteration 4	ONE_BY_ONE	✗	2	122,42	15885	6305	\$ 0,095	5,12	237	49	22,75	4,75	1	1,75	73,54	✗
Iteration 5	ONE_BY_ONE	✗	2	122,82	15530	5662	\$ 0,088	7,18	220	47	24,29	5,14	1	1,86	74,29	✗
Average	ONE_BY_ONE	100%	2,0	121,52	15791	6117	\$ 0,093	7,38	234,8	48,8	24,69	5,18	1	1,82	73,78	100%

3.2.2 Fallstudie 2

Die zweite Fallstudie beschäftigt sich mit einem Python-Code, der typische Fehler, sogenannte Code-Smells, eines fortgeschrittenen Entwicklers berücksichtigt:

Erfahrungsgrad	Anzahl an Regelverstößen	Code-Smells
Intermediate	8	<ul style="list-style-type: none"> Mäßige Anzahl an redundanten Kommentaren Inkorrekter Import in einer Zeile Imports in falscher Reihenfolge Inkonsistente Groß- und Kleinschreibung Keine Verwendung von Type Hints Keine Verwendung von Docstrings Keine Verwendung des Main-Name-Idioms Keine Verwendung von Logging

Programmcode vor der Optimierung (siehe Anhang F: case_study_2.py):

Tabelle 16: Auswertung der Fallstudie 2 vor der Optimierung

LOC	SLOC	Commented Lines	Average function length		Average CC	MI	Pylint Score
			with Docstrings	without Docstrings			
87	82	6	15.2	15.2	3.40 (A)	42.42 (A)	7.10

Programmcode nach der Optimierung mit dem „All-at-Once“-Ansatz:

	Strategy	Reliability	Rule violations	Optimization time [s]	Token usage	PyLint Score	LOC	SLOC	Avg. Func. Len.	Commented Lines	Avg. CC	MIL	Functional Correctness	Notable Findings
				Input	Output		with Docstrings	without Docstrings						
Iteration 1	All_AT_ONCE	✗	2	19,17	13249	1458	\$ 0,062	9,76	199	92	17,56	9,00	2	2,33 69,98 ✗
Iteration 2	All_AT_ONCE	✗	2	18,08	13249	1458	\$ 0,062	9,76	199	92	17,56	9,00	2	2,33 69,98 ✗
Iteration 3	All_AT_ONCE	✗	2	18,10	13249	1458	\$ 0,062	9,76	199	92	17,56	9,00	2	2,33 69,98 ✗
Iteration 4	All_AT_ONCE	✗	2	18,24	13249	1458	\$ 0,062	9,76	199	92	17,56	9,00	2	2,33 69,98 ✗
Iteration 5	All_AT_ONCE	✗	2	18,25	13249	1458	\$ 0,062	9,76	199	92	17,56	9,00	2	2,33 69,98 ✗
Average	All_AT_ONCE	100%	2,0	18,37	13249	1458	\$ 0,062	9,76	199	92,0	17,56	9,00	2,0	2,33 69,98 100%
Iteration 1	All_AT_ONCE	✗	1	14,98	12704	1805	\$ 0,025	10,00	262	91	21,00	8,70	3	2,20 68,32 ✗
Iteration 2	All_AT_ONCE	✗	1	15,60	12704	1819	\$ 0,025	10,00	254	90	20,00	8,60	3	2,20 68,43 ✗
Iteration 3	All_AT_ONCE	✗	1	15,65	12704	1993	\$ 0,026	10,00	290	99	19,42	8,00	3	2,08 67,90 ✗
Iteration 4	All_AT_ONCE	✗	1	13,74	12704	1750	\$ 0,025	10,00	234	88	18,30	8,40	3	2,20 68,64 ✗
Iteration 5	All_AT_ONCE	✗	0	16,93	12704	2021	\$ 0,026	10,00	270	96	26,38	10,38	3	2,62 67,5 ✗
Average	All_AT_ONCE	100%	0,8	15,38	12704	1878	\$ 0,025	10,00	262,0	92,8	21,02	8,82	3,0	2,26 68,40 100%
Iteration 1	All_AT_ONCE	✗	2	25,33	11163	933	\$ 0,037	9,85	141	75	23,80	14,20	2	3,40 71,56 ✗
Iteration 2	All_AT_ONCE	✗	2	24,19	11163	952	\$ 0,037	9,85	142	76	24,00	14,40	2	3,40 70,55 ✗
Iteration 3	All_AT_ONCE	✗	2	28,06	11163	929	\$ 0,037	9,85	141	75	23,80	14,20	2	3,40 71,56 ✗
Iteration 4	All_AT_ONCE	✗	2	25,32	11163	941	\$ 0,037	9,85	142	75	23,80	14,20	2	3,40 70,9 ✗
Iteration 5	All_AT_ONCE	✗	2	30,72	11163	938	\$ 0,037	9,85	141	75	23,80	14,20	2	3,40 71,19 ✗
Average	All_AT_ONCE	100%	2,0	26,32	11163	938,6	\$ 0,037	9,85	141,4	75,2	23,84	14,24	2,0	3,40 71,15 100%
Iteration 1	All_AT_ONCE	✗	3	23,75	11263	1274	\$ 0,035	8,73	208	78	26,33	11,83	2	3,00 69,18 ✗
Iteration 2	All_AT_ONCE	✗	3	23,27	11263	1253	\$ 0,035	8,26	208	76	26,00	11,50	2	2,83 70,85 ✗
Iteration 3	All_AT_ONCE	✗	3	24,52	11263	1337	\$ 0,036	8,63	214	82	27,00	12,50	2	3,17 67,80 ✗
Iteration 4	All_AT_ONCE	✗	3	22,9	11263	1198	\$ 0,035	9,55	200	73	26,00	11,50	0	2,83 71,23 ✗
Iteration 5	All_AT_ONCE	✗	3	25,31	11263	1246	\$ 0,035	8,35	208	76	26,00	11,50	2	2,83 70,85 ✗
Average	All_AT_ONCE	100%	3,0	23,95	11263	1262	\$ 0,035	8,74	207,6	77,0	26,27	11,77	1,6	2,93 69,94 100%
Iteration 1	All_AT_ONCE	✗	3	24,44	11263	1274	\$ 0,035	8,73	208	78	26,33	11,83	2	3,00 69,18 ✗
Iteration 2	All_AT_ONCE	✗	3	24,44	11263	1274	\$ 0,035	8,73	208	78	26,33	11,83	2	3,00 69,18 ✗
Iteration 3	All_AT_ONCE	✗	3	24,44	11263	1274	\$ 0,035	8,73	208	78	26,33	11,83	2	3,00 69,18 ✗
Iteration 4	All_AT_ONCE	✗	3	24,44	11263	1274	\$ 0,035	8,73	208	78	26,33	11,83	2	3,00 69,18 ✗
Iteration 5	All_AT_ONCE	✗	3	24,44	11263	1274	\$ 0,035	8,73	208	78	26,33	11,83	2	3,00 69,18 ✗
Average	All_AT_ONCE	100%	3,0	24,44	11263	1274	\$ 0,035	8,73	208	78	26,33	11,83	2	3,00 69,18 ✗

Programmcode nach der Optimierung mit dem „One-by-One“-Ansatz

	Strategy	Reliability	Rule violations	Optimization time [s]	Token usage		Optimization costs	Pylint Score	LOC	SLOC	Avg. Func. Len.		Commented Lines	Avg. CC	MI	Functional Correctness	Notable Findings
					Input	Output					with Docstrings	without Docstrings					
Iteration 1	ONE_BY_ONE	☒	1	136.94	23199	12098	\$ 0.251	9,78	379	127	22.00	8.29	3	1.86	63,71	☒	
Iteration 2	ONE_BY_ONE	☒	1	139.75	23251	12202	\$ 0.253	9,56	383	127	22.29	8.29	3	1.86	63,71	☒	
Iteration 3	ONE_BY_ONE	☒	1	139.73	23255	12204	\$ 0.253	9,56	379	123	22.00	8.00	3	1.86	64,01	☒	Max line length exceeded
Iteration 4	ONE_BY_ONE	☒	1	140.37	23262	12202	\$ 0.253	9,56	375	118	21.71	7.64	3	1.86	64,4	☒	
Iteration 5	ONE_BY_ONE	☒	1	146.57	23199	12098	\$ 0.251	9,78	379	127	22.00	8.29	3	1.86	63,71	☒	
Average	ONE_BY_ONE	100%	1,0	140.67	23233	12161	\$ 0.252	9,65	379	124,4	22.00	8,10	3,0	1.86	63,91	100%	
Iteration 1	ONE_BY_ONE	☒	0	124,2	22560	12542	\$ 0.081	9,88	426	95	32.18	7.55	0	2.00	68,28	☒	
Iteration 2	ONE_BY_ONE	☒	0	116,59	22014	11450	\$ 0.085	10,00	405	94	31,09	7,64	1	2.09	68,03	☒	
Iteration 3	ONE_BY_ONE	☒	0	124,26	23364	13499	\$ 0.097	10,00	462	101	35,45	7,91	0	2.09	67,19	☒	(Imports not commented)
Iteration 4	ONE_BY_ONE	☒	0	109,4	21521	10451	\$ 0.079	10,00	327	94	23,64	7,64	0	2.09	68,03	☒	
Iteration 5	ONE_BY_ONE	☒	0	138	23517	14358	\$ 0,101	9,19	539	115	38,58	8,42	0	2.00	66,53	☒	
Average	ONE_BY_ONE	100%	0,0	122,49	22595	12460	\$ 0,091	9,81	431,8	99,8	32,19	7,83	0,2	2,05	67,61	100%	
Iteration 1	ONE_BY_ONE	☒	0	165,9	17192	7794	\$ 0,121	10,00	213	82	21,25	9,75	3	2,50	69,78	☒	
Iteration 2	ONE_BY_ONE	☒	0	174,52	17631	8184	\$ 0,126	10,00	223	95	19,00	9,00	3	2,33	67,9	☒	
Iteration 3	ONE_BY_ONE	☒	0	210,63	17780	8508	\$ 0,130	10,00	241	93	19,78	8,67	3	2,44	67,83	☒	(Deep Nesting)
Iteration 4	ONE_BY_ONE	☒	0	184,07	17979	8842	\$ 0,133	10,00	253	98	22,33	9,33	3	2,44	66,96	☒	
Iteration 5	ONE_BY_ONE	☒	0	182,15	17544	8341	\$ 0,127	10,00	247	91	22,78	9,56	3	2,44	67,66	☒	
Average	ONE_BY_ONE	100%	0,0	183,45	17625	8333,8	\$ 0,127	10,00	235,4	91,8	21,03	9,26	3,0	2,43	68,03	100%	
Iteration 1	ONE_BY_ONE	☒	2	182,96	18887	10249	\$ 0,140	7,93	354	96	27,00	7,55	1	2,09	66,26	☒	
Iteration 2	ONE_BY_ONE	☒	2	183,62	19142	10532	\$ 0,144	8,12	361	100	27,64	8,00	1	2,09	65,67	☒	
Iteration 3	ONE_BY_ONE	☒	2	185,44	19266	10780	\$ 0,146	8,35	387	106	26,92	7,75	3	2,17	63,95	☒	Max line length exceeded Deleted pylet import Imported pyglet within function
Iteration 4	ONE_BY_ONE	☒	2	181,25	19124	10723	\$ 0,145	7,34	377	90	29,64	7,27	0	2,09	67,08	☒	
Iteration 5	ONE_BY_ONE	☒	2	177,19	18992	10373	\$ 0,142	9,11	361	102	27,91	8,55	1	2,09	65,68	☒	
Average	ONE_BY_ONE	100%	2,0	182,09	19082	10531	\$ 0,143	8,17	388,0	98,8	27,82	7,82	1,2	2,11	65,73	40%	

3.2.3 Fallstudie 3

Die dritte Fallstudie beschäftigt sich mit einem Python-Code, der typische Fehler, sogenannte Code-Smells, eines professionellen Entwicklers berücksichtigt:

Erfahrungsgrad	Anzahl an Regelverstößen	Code-Smells
Professional	3	<ul style="list-style-type: none"> • Falsche Reihenfolge der Imports • Keine Verwendung von Docstrings • Keine Verwendung von Logging

Programmcode vor der Optimierung (siehe Anhang G: case_study_3.py):

Tabelle 17: Auswertung der Fallstudie 3 vor der Optimierung

LOC	SLOC	Commented Lines	Average function length		Average CC	MI	Pylint Score
			with Docstrings	without Docstrings			
331	275	1	22.67	22.67	2.56 (A)	24.79 (A)	8.94

Programmcode nach der Optimierung mit dem „All-at-Once“-Ansatz:

	Strategy	Reliability	Rule violations	Optimization time [s]		Token usage	Optimization costs	Pylint Score	LOC	SLOC	Avg. Func. Len.		Commented Lines	Avg. CC	MI	Functional Correctness	Notable Findings
				Input	Output						with Docstrings	without Docstrings					
Iteration 1	ALL_AT_ONCE	✗	1	68,53	161,38	5678	\$ 0,134	10,00	631	277	37,42	22,83	3	2,56	50,72	✗	
Iteration 2	ALL_AT_ONCE	✗	1	66,19	161,38	5546	\$ 0,132	10,00	611	277	35,92	22,83	3	2,56	50,72	✗	
Iteration 3	ALL_AT_ONCE	✗	1	67,01	161,38	5584	\$ 0,132	10,00	613	277	35,92	22,83	3	2,56	50,72	✗	No Logging
Iteration 4	ALL_AT_ONCE	✗	1	68,01	161,38	5587	\$ 0,132	10,00	614	277	36,00	22,83	3	2,56	50,72	✗	
Iteration 5	ALL_AT_ONCE	✗	1	67,17	161,38	5584	\$ 0,132	10,00	614	277	36,00	22,83	3	2,56	50,72	✗	
Average	ALL_AT_ONCE	100%	1,0	67,38	161,38	5595,8	\$ 0,132	10,00	616,6	277,0	36,25	22,83	3,0	2,56	50,72	100%	
Iteration 1	ALL_AT_ONCE	✗	4	44,5	154,74	5889	\$ 0,049	9,42	635	332	39,75	27,33	1	2,56	49,62	✗	
Iteration 2	ALL_AT_ONCE	✗	3	43,89	154,74	5862	\$ 0,049	9,94	634	291	35,85	22,23	1	2,47	50,99	✗	
Iteration 3	ALL_AT_ONCE	✗	3	38,48	154,74	5377	\$ 0,046	9,71	589	286	35,58	23,67	3	2,56	51,18	✗	
Iteration 4	ALL_AT_ONCE	✗	3	47,29	154,74	6248	\$ 0,051	9,57	649	309	33,64	21,14	3	2,39	50,25	✗	
Iteration 5	ALL_AT_ONCE	✗	3	44,1	154,74	6056	\$ 0,050	9,94	638	298	39,08	24,67	1	2,56	50,64	✗	
Average	ALL_AT_ONCE	100%	3,2	43,65	154,74	5886	\$ 0,049	9,72	629,0	303,2	36,78	23,81	1,8	2,51	50,54	80%	
Iteration 1	ALL_AT_ONCE	✗	2	96,66	13261	4127	\$ 0,074	9,18	344	281	23,25	23,17	4	2,56	38,52	✗	
Iteration 2	ALL_AT_ONCE	✗	1	102,31	13261	4127	\$ 0,074	10,00	601	277	34,92	22,83	3	2,56	50,72	✗	
Iteration 3	ALL_AT_ONCE	✗	1	128,93	13261	4139	\$ 0,075	10,00	603	277	34,92	22,83	3	2,56	50,72	✗	
Iteration 4	ALL_AT_ONCE	✗	1	102,27	13261	3898	\$ 0,072	9,82	548	277	34,92	22,83	3	2,56	52,77	✗	
Iteration 5	ALL_AT_ONCE	✗	1	98,74	13261	4098	\$ 0,074	10,00	601	277	34,92	22,83	3	2,56	50,72	✗	
Average	ALL_AT_ONCE	100%	1,2	105,78	13261	4077,8	\$ 0,074	9,80	539,4	277,8	32,59	22,90	3,2	2,56	48,69	100%	
Iteration 1	ALL_AT_ONCE	✗	3	85,18	13411	4041	\$ 0,067	9,59	549	278	35,00	22,92	1	2,56	52,74	✗	
Iteration 2	ALL_AT_ONCE	✗	3	87,35	13411	4230	\$ 0,069	9,31	568	280	31,57	19,71	1	2,39	52,52	✗	
Iteration 3	ALL_AT_ONCE	✗	4	77,65	13411	3889	\$ 0,067	8,18	532	278	38,00	22,92	1	2,56	53,23	✗	
Iteration 4	ALL_AT_ONCE	✗	4	89,32	13411	4587	\$ 0,073	8,41	592	280	35,17	23,08	1	2,56	50,4	✗	
Iteration 5	ALL_AT_ONCE	✗	4	96,35	13411	4452	\$ 0,071	8,88	597	280	35,17	23,08	1	2,56	50,4	✗	
Average	ALL_AT_ONCE	100%	3,6	87,17	13411	4260	\$ 0,069	8,87	567,6	279,2	34,98	22,34	1,0	2,33	51,86	40%	

Programmcode nach der Optimierung mit dem „One-by-One“-Ansatz

	Strategy	Reliability	Rule violations	Optimization time [s]	Token usage		Optimization costs	Pylint Score	LOC	SLOC	with Docstrings	without Docstrings	Avg. Func. Len.	Commented Lines	Avg. CC	MI	Functional Correctness	Notable Findings
					Input	Output												
Iteration 1	ONE_BY_ONE	✗	0	467,08	48895	39308	\$ 0,736	9,69	793	329	28,10	14,76	3	1,96	49,13	✗		
Iteration 2	ONE_BY_ONE	✗	0	462,17	48278	38488	\$ 0,722	9,74	794	324	26,55	13,77	3	1,92	49,28	✗		
Iteration 3	ONE_BY_ONE	✗	0	461,24	48317	38555	\$ 0,723	9,69	776	321	27,38	14,38	2	1,96	49,37	✗		
Iteration 4	ONE_BY_ONE	✗	0	460,81	48275	38482	\$ 0,722	9,74	794	324	26,55	13,77	3	1,92	49,28	✗		
Iteration 5	ONE_BY_ONE	✗	0	461,63	48397	38717	\$ 0,726	9,74	801	324	26,86	13,77	2	1,92	49,28	✗		
Average	ONE_BY_ONE	100%	0,0	462,59	48432	38710	\$ 0,726	9,72	791,6	324,4	27,09	14,09	2,6	1,94	49,27	100%		
Iteration 1	ONE_BY_ONE	✗	4	271,45	47751	38869	\$ 0,254	9,88	827	384	36,88	21,71	1	2,29	47,33	✗		
Iteration 2	ONE_BY_ONE	✗	4	273,44	47673	38834	\$ 0,254	9,94	840	359	33,53	18,11	2	2,00	48,69	✗		
Iteration 3	ONE_BY_ONE	✗	4	270,54	47570	38760	\$ 0,253	9,94	825	379	35,76	21,18	2	2,10	47,47	✗		
Iteration 4	ONE_BY_ONE	✗	3	264,5	47694	38713	\$ 0,253	9,31	833	380	36,29	21,29	0	2,24	47,22	✗		
Iteration 5	ONE_BY_ONE	✗	4	268,91	46898	37593	\$ 0,247	9,82	794	382	36,18	21,41	2	2,05	48,02	✗		
Average	ONE_BY_ONE	100%	3,8	269,77	47517	38554	\$ 0,252	9,78	823,8	376,8	35,73	20,74	1,4	2,14	47,75	100%		
Iteration 1	ONE_BY_ONE	✗	0	468,79	34709	24919	\$ 0,336	9,83	558	286	35,00	22,83	3	2,56	52,47	✗		
Iteration 2	ONE_BY_ONE	✗	0	471,66	34782	24826	\$ 0,335	9,83	554	286	34,50	22,83	3	2,56	52,42	✗		
Iteration 3	ONE_BY_ONE	✗	0	493,12	34882	24943	\$ 0,337	9,83	572	286	35,50	22,83	3	2,56	52,42	✗		
Iteration 4	ONE_BY_ONE	✗	0	559,85	34988	25181	\$ 0,339	9,83	566	286	35,42	22,83	3	2,56	52,47	✗		
Iteration 5	ONE_BY_ONE	✗	0	624,77	34795	24897	\$ 0,336	9,83	563	286	35,00	22,83	3	2,56	52,47	✗		
Average	ONE_BY_ONE	100%	0,0	523,64	34831	24953	\$ 0,337	9,83	562,6	286	35,08	22,83	3,0	2,56	52,45	100%		
Iteration 1	ONE_BY_ONE	✗	CRASHED				\$ -											
Iteration 2	ONE_BY_ONE	✗	CRASHED				\$ -											
Iteration 3	ONE_BY_ONE	✗	CRASHED				\$ -											
Iteration 4	ONE_BY_ONE	✗	CRASHED				\$ -											
Iteration 5	ONE_BY_ONE	✗	CRASHED				\$ -											
Average	ONE_BY_ONE	0%	0,0	0,00	0,0	0	\$ -	0,00	0,0	0,0	0,00	0,0	0,0	0,0	0,00	0,00	0%	

4 Ergebnis und Diskussion

4.1 Ergebnis

4.1.1 Forschungsfrage 1

Welche konkreten Maßnahmen können in Python getroffen werden, um die innere Softwarequalität zu verbessern, und wie lassen sich diese Maßnahmen in einem praxisnahen Clean-Code Leitfaden strukturiert zusammenfassen?

Um die innere Softwarequalität in Python zu verbessern, können verschiedene Maßnahmen ergriffen werden. Dazu zählt die Verwendung pythonspezifischer, integrierter, optimierter Funktionen (z.B.: List Comprehensions, Context-Manager, f-Strings), das korrekte Importieren von Modulen, die Einhaltung von Namenskonvention gemäß *PEP-8*, die Ergänzung von Typannotationen gemäß *PEP-484*, der Einsatz von Docstrings für Klassen, Methoden, Funktionen und Modulen nach *PEP-257* und *PEP-287*, die Berücksichtigung von Formatvorgaben gemäß *PEP-8*, sowie die Verwendung einer *main()*-Funktion in Kombination mit dem Main-Name-Idiom (*if __name__ == "__main__":*).

Sämtliche Clean-Code-Maßnahmen wurden praxisnah anhand von Beispielen im WRONG- und CORRECT-Format erläutert und strukturiert in einer Markdown-Datei dokumentiert. Dabei wurde hoher Wert auf die modulare Anpassbarkeit und Erweiterbarkeit der einzelnen Kategorien gelegt. Dies ist einerseits essenziell für eine potenzielle spätere Neu-Anordnung der einzelnen Abschnitte und andererseits für die Sicherstellung einer flexiblen und skalierbaren Struktur. Des Weiteren lassen sich Markdown-Dateien einfach rendern, wodurch sowohl die Analyse als auch die Lesbarkeit für Menschen verbessert wird. Darüber hinaus empfiehlt sich die Verwendung einer Markdown-Datei für die Verarbeitung des Leitfadens innerhalb eines LLMs [94].

Zusammengefasst lässt sich der Leitfaden wie folgt quantifizieren:

- 8 Kategorien
- 52 Regeln
- 74 Beispiele
- 1658 Zeilen
- 45878 Zeichen

Eine detaillierte Erläuterung sämtlicher Maßnahmen befindet sich sowohl im Clean-Code-Leitfaden für die Python-Entwicklung (auf Deutsch) als auch im Anhang A: guideline.md (auf Englisch).

4.1.2 Forschungsfrage 2

Wie kann die automatisierte Anwendung des entwickelten Clean-Code Leitfadens durch ein Large Language Model technisch umgesetzt werden, und inwiefern ist ein solcher Ansatz praktikabel bzw. wirtschaftlich sinnvoll?

Technische Umsetzung:

Die automatisierte Anwendung des entwickelten Clean-Code-Leitfadens kann technisch mithilfe von zwei Ansätzen umgesetzt werden:

- „All-at-Once“

Hier wird der gesamte Leitfaden in einem Durchlauf angewendet.

- „One-by-One“

Hier wird der Leitfaden in mehrere Kapitel aufgeteilt und in acht Durchläufen iterativ angewendet.

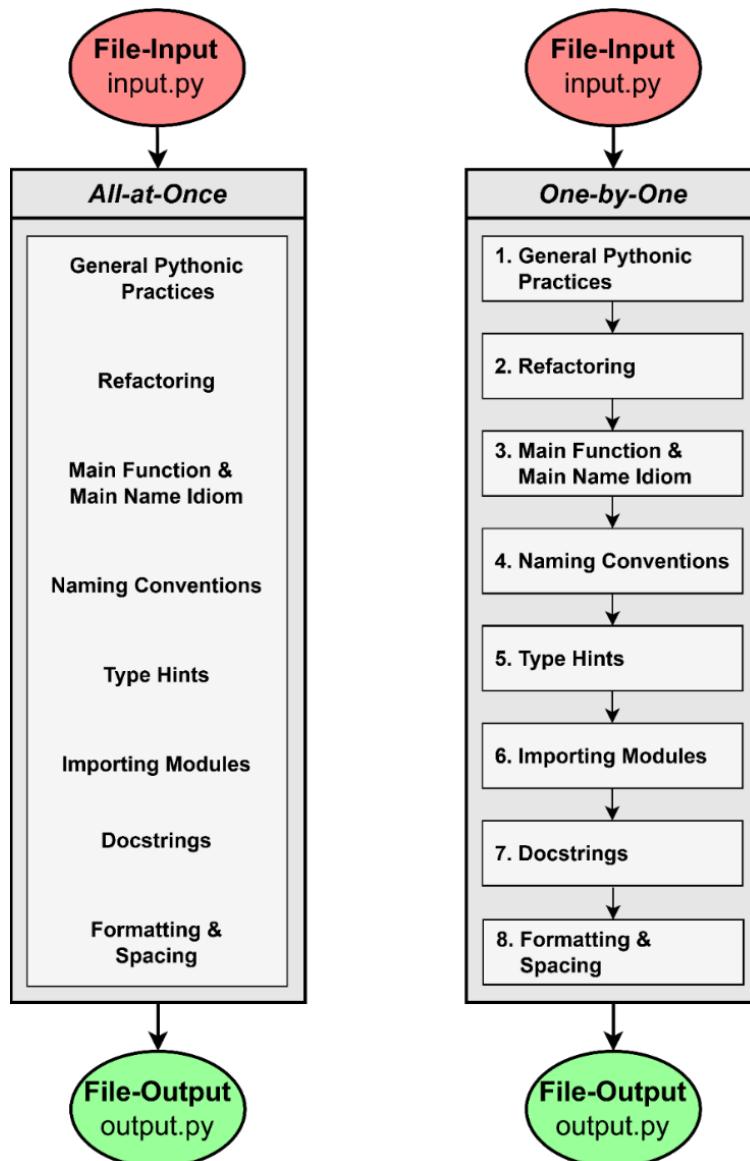


Abbildung 43: Ablaufdiagramm des "All-at-Once"- und des "One-by-One"-Ansatzes

Ein entscheidendes Kriterium bei der Modellauswahl war die Möglichkeit, die Temperatur anzupassen. Die Definition der Temperatur gleich Null ist essenziell, um deterministische und reproduzierbare Ergebnisse zu gewährleisten.

Derzeit ist die Nutzung der Modelle von folgenden vier LLM-Anbietern möglich:

- Anthropic (z.B.: claude-3-7-sonnet-20250219)
- Gemini (z.B.: gemini-2.0-pro-exp-02-05)
- OpenAI (z.B.: gpt-4o-2024-08-06)
- Xai (z.B.: grok-2-1212)

Für unerfahrene bzw. fortgeschrittene Softwareentwickler:innen wird der Einsatz des gpt-4o-2024-08-06 Modells von OpenAI in Kombination mit dem „One-by-One“-Ansatz empfohlen. Im Gegensatz dazu profitieren professionelle Softwareentwickler:innen von einer bedarfsgerechten Komprimierung des Clean-Code-Leitfadens in Kombination mit dem „All-at-Once“-Ansatz, um die Effizienz zu verbessern sowie die Kosten und Dauer der Optimierung zu minimieren.

Das entwickelte Tool nennt sich OptiPy und steht in drei Varianten zur Verfügung:

1. Library Integration

→ Einbindung als Bibliothek in bestehende Python-Anwendungen.

```
from pathlib import Path
from optipy import Optipy

optimizer = Optipy(
    filepath=Path("example.py"),
    strategy="one_by_one",
    model="gpt-4o",
    debug=True
)
optimizer.start()
```

2. Command-Line Interface Tool

→ Direkte Nutzung über die Kommandozeile.

```
git clone https://github.com/phgas/optipy.git
cd optipy
python optipy.py --filepath=example.py --strategy=one_by_one --model=gpt-4o --debug
```

3. CI/CD-Pipeline

→ Automatisierte Einbindung in Entwicklungs- und Deployment-Prozesse, beispielsweise GitHub Actions. (siehe Anhang D: optipy-workflow.yml)

Hinterfragung der Praktikabilität:

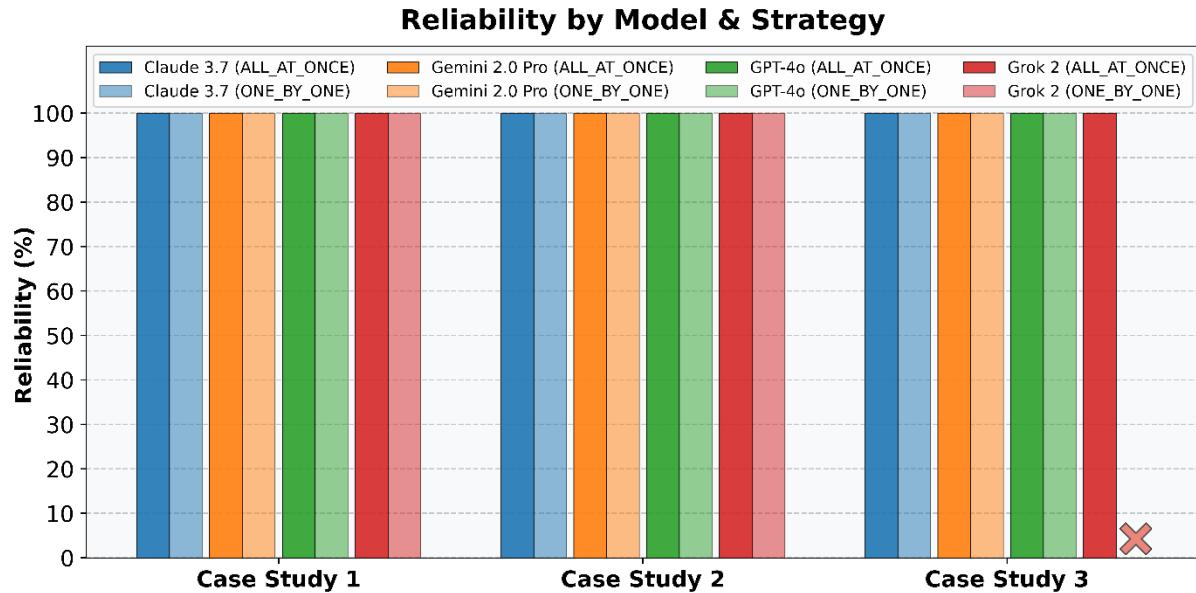


Abbildung 44: Vergleich der Zuverlässigkeit verschiedener LLMs und Strategien über drei Fallstudien hinweg

Die Zuverlässigkeit hebt hervor, dass die Verwendung von grok-2-1212 im Kontext dieser Arbeit nicht empfehlenswert ist.

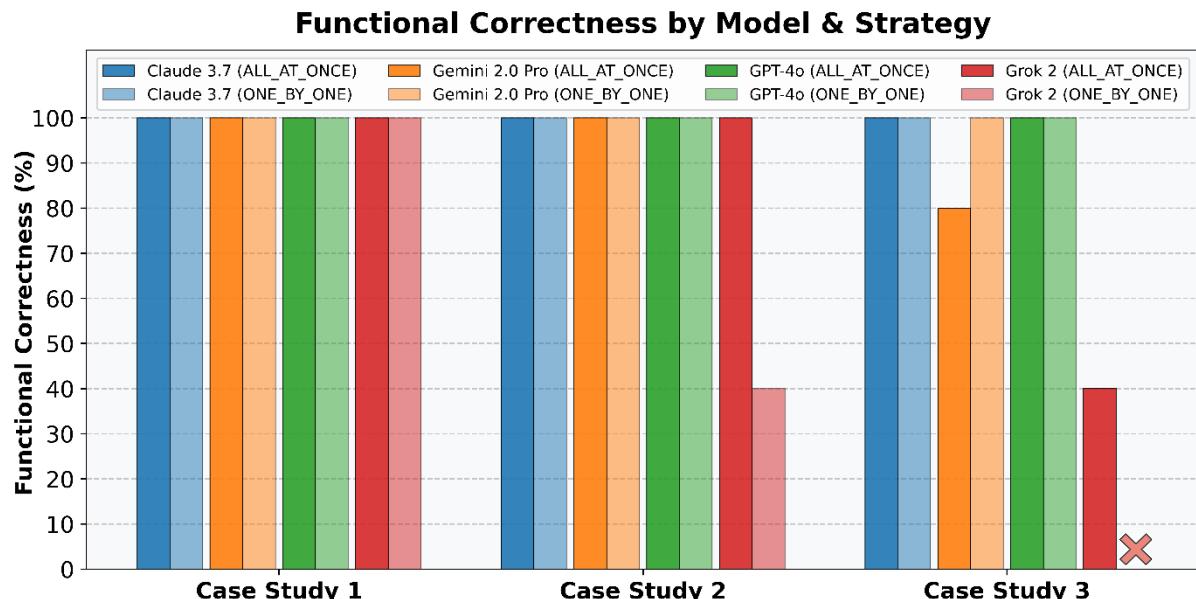


Abbildung 45: Vergleich der funktionalen Korrektheit verschiedener LLMs und Strategien über drei Fallstudien hinweg

Die funktionale Korrektheit überprüft, ob die ursprüngliche Funktionalität beibehalten wurde. Die Ergebnisse heben hervor, dass lediglich claude-3-7-sonnet-20250219 und gpt-4o-2024-08-06 in allen Fallstudien mit beiden Optimierungsansätzen eine durchgängig konsistente Leistungsfähigkeit aufweisen.

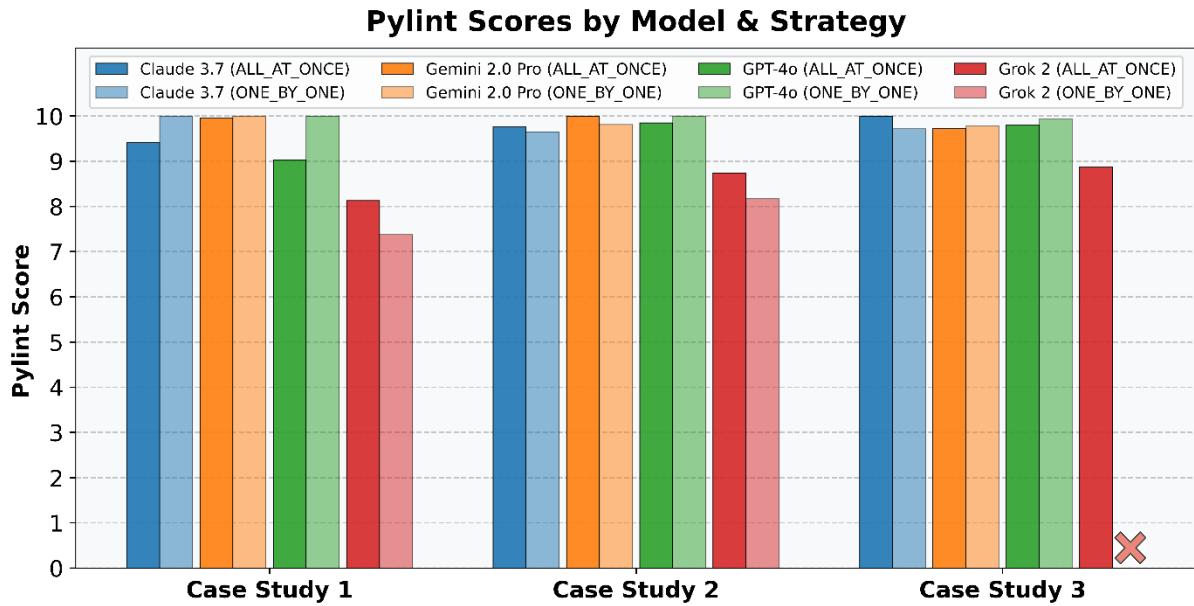


Abbildung 46: Vergleich der Pylint-Scores verschiedener LLMs und Strategien über drei Fallstudien hinweg

Die Pylint-Scores von claude-3-7-sonnet-20250219, gemini-2.0-pro-exp-02-05 und gpt-4o-2024-08-06 befinden sich konsistent im Bereich von 9 bis 10 über alle Fallstudien hinweg. Im Vergleich dazu schneidet grok-2-1212 signifikant schlechter ab.

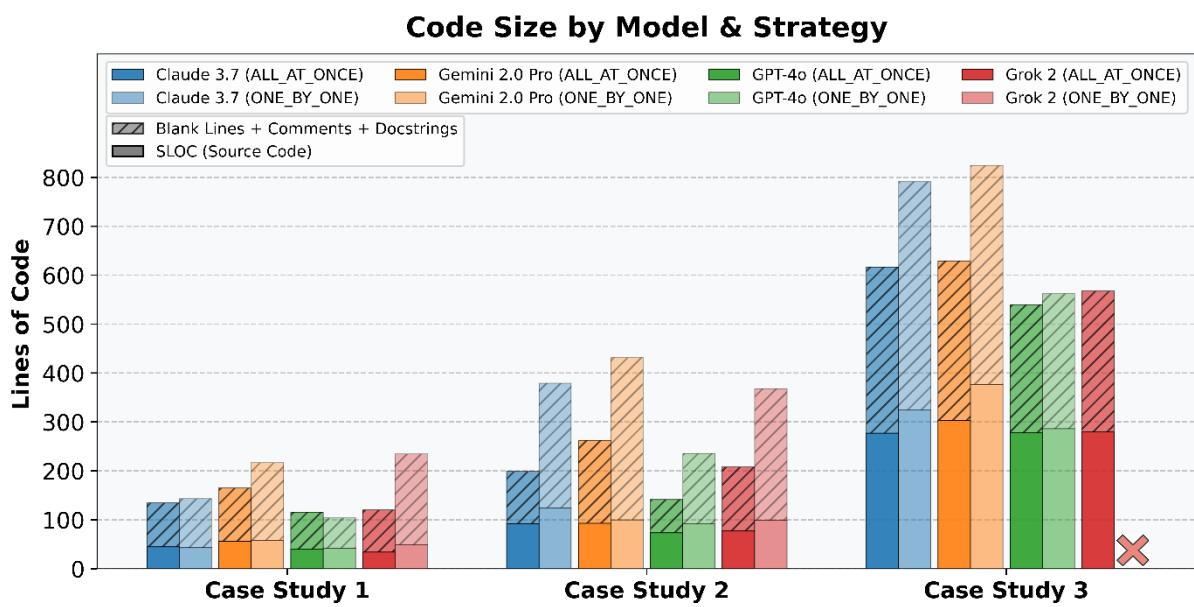


Abbildung 47: Vergleich der Zeilenanzahl verschiedener LLMs und Strategien über drei Fallstudien hinweg

Der „One-by-One“-Ansatz führt in den meisten Fällen im Vergleich zum „All-at-Once“-Ansatz zu einer detaillierteren und umfangreicheren Codegenerierung. Dies ist hauptsächlich auf die vermehrte Nutzung von Docstrings zurückzuführen. Eine weitere interessante Beobachtung ist, dass gpt-4o-2024-08-06 im Durchschnitt die kompakteste Lösung erstellt.

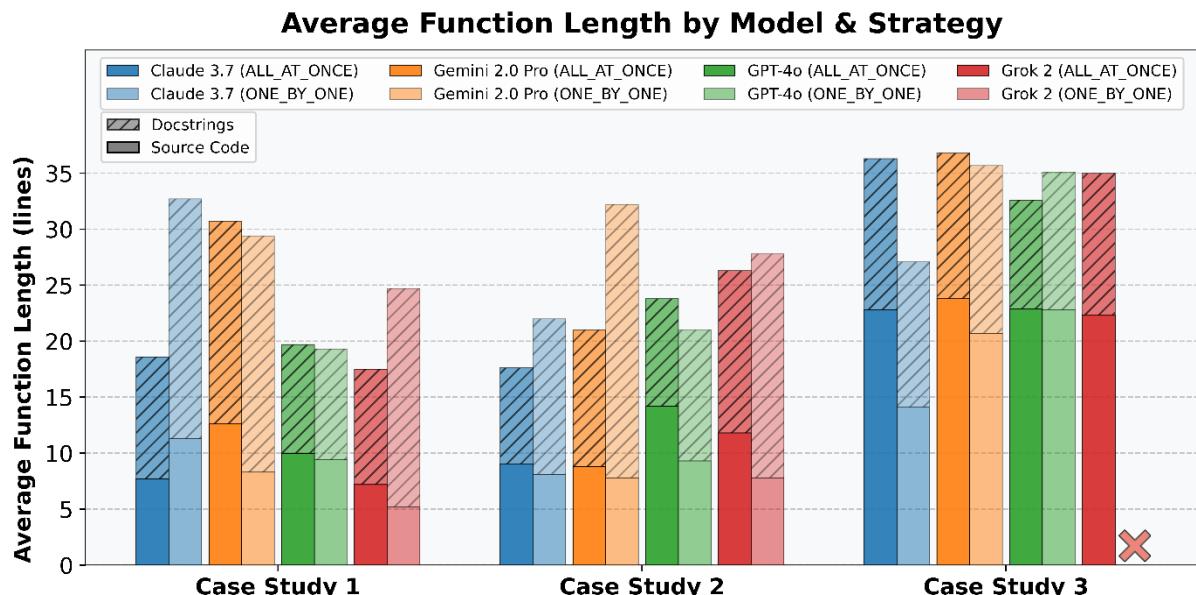


Abbildung 48: Vergleich der durchschnittlichen Funktionslänge (mit und ohne Docstrings) verschiedener LLMs und Strategien über drei Fallstudien hinweg

Abbildung 48 deutet darauf hin, dass der „One-by-One“-Ansatz den durchschnittlichen Anteil an ausführbarem Code (SLOC) reduzieren konnte, was auf ein erfolgreiches Refactoring mit kürzeren, wartungsfreundlichen Funktionen hindeutet. Gleichzeitig führt dieser Ansatz jedoch in den Fallstudien 1 und 2 zu einer ausführlicheren – teilweise möglicherweise übermäßig detaillierten – Dokumentation durch Docstrings (eine Ausnahme bildet gpt-4o-2024-08-06). In der dritten Fallstudie sticht claude-3-7-sonnet-20250219 unter Verwendung des „One-by-One“-Ansatzes mit einem im Vergleich zu den anderen Modellen geringeren SLOC-Anteil von 14.1 hervor. Dies deutet drauf hin, dass dieses Modell insbesondere für professionelle Softwareentwickler:innen vorteilhaft sein könnte.

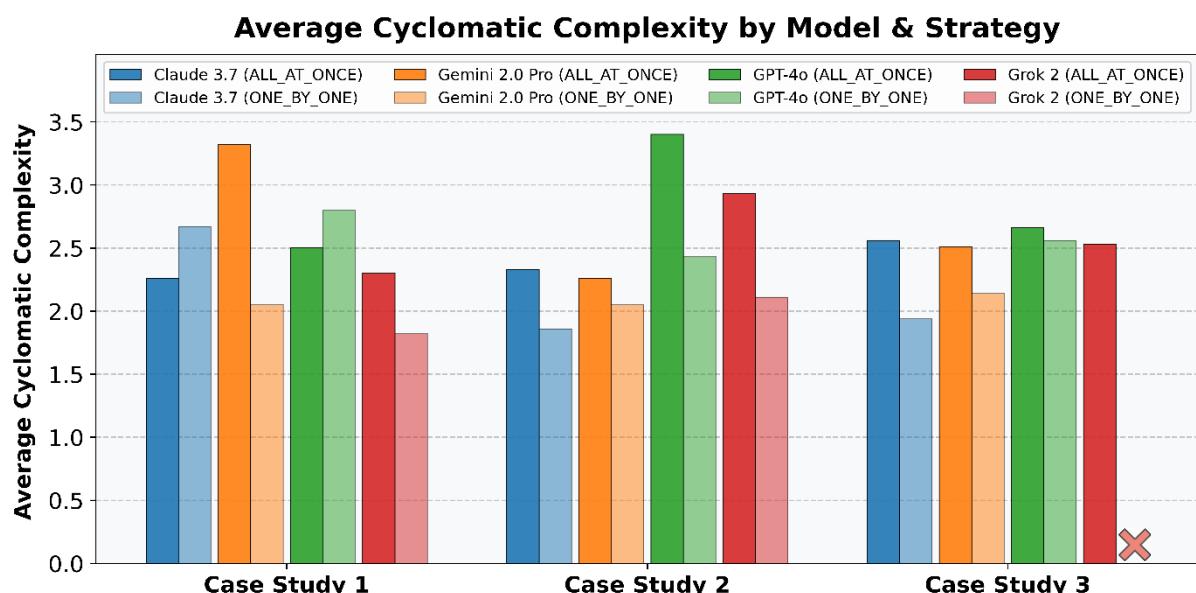


Abbildung 49: Vergleich der durchschnittlichen zyklomatischen Komplexität verschiedener LLMs und Strategien über drei Fallstudien hinweg

In allen Fallstudien konnte gewährleistet werden, dass die durchschnittliche zyklomatische Komplexität einen Grenzwert von 6 unterschreitet.

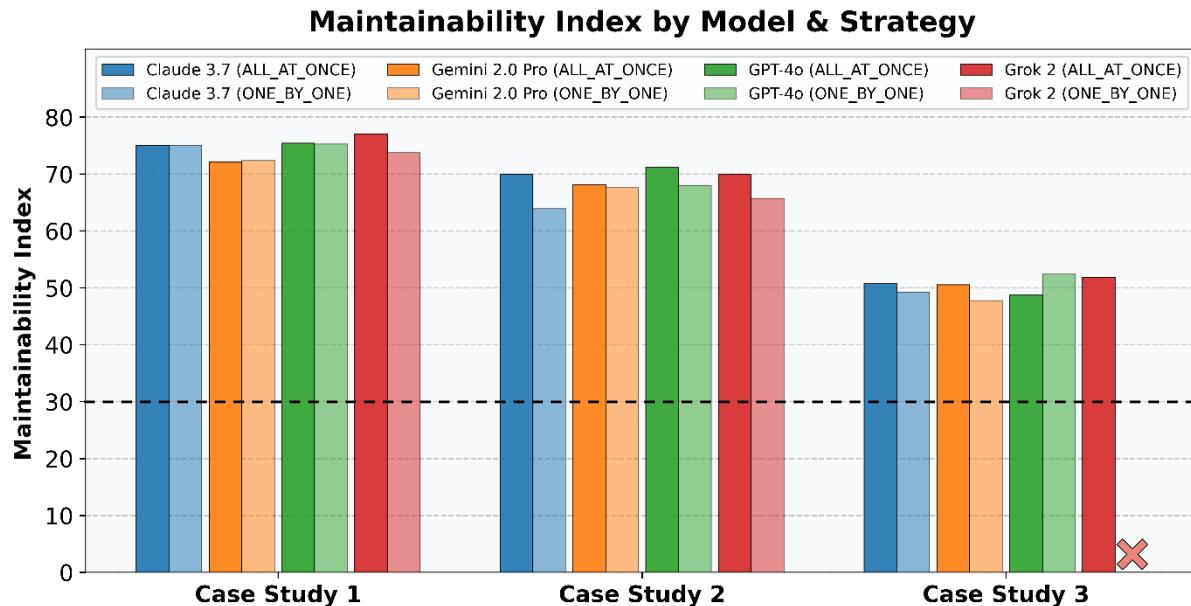


Abbildung 50: Vergleich des Maintainability-Indexes verschiedener LLMs und Strategien über drei Fallstudien hinweg

In allen Fallstudien konnte sichergestellt werden, dass die Wartbarkeit einen Grenzwert von 30 überschreitet. Des Weiteren konnte die Kritik, die am Maintainability Index geäußert wurde, im Rahmen dieser Arbeit bestätigt werden. Obwohl die Qualität (kaum Regelverstöße) des Programmcodes in der dritten Fallstudie am höchsten ist, weist sie den geringsten Wert auf.

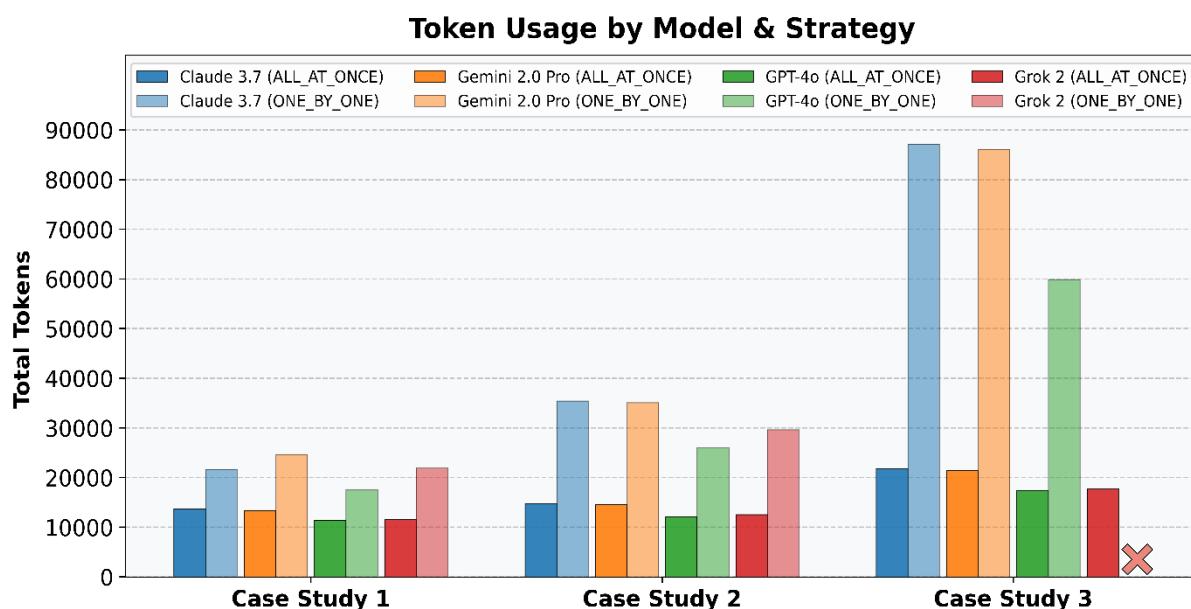


Abbildung 51: Vergleich des Tokenverbrauchs verschiedener LLMs und Strategien über drei Fallstudien hinweg

Bei dem Tokenverbrauch handelt es sich um die Summe aus Eingabe- und Ausgabe-Tokens, die in dem jeweiligen Optimierungsprozess verwendet wurden.

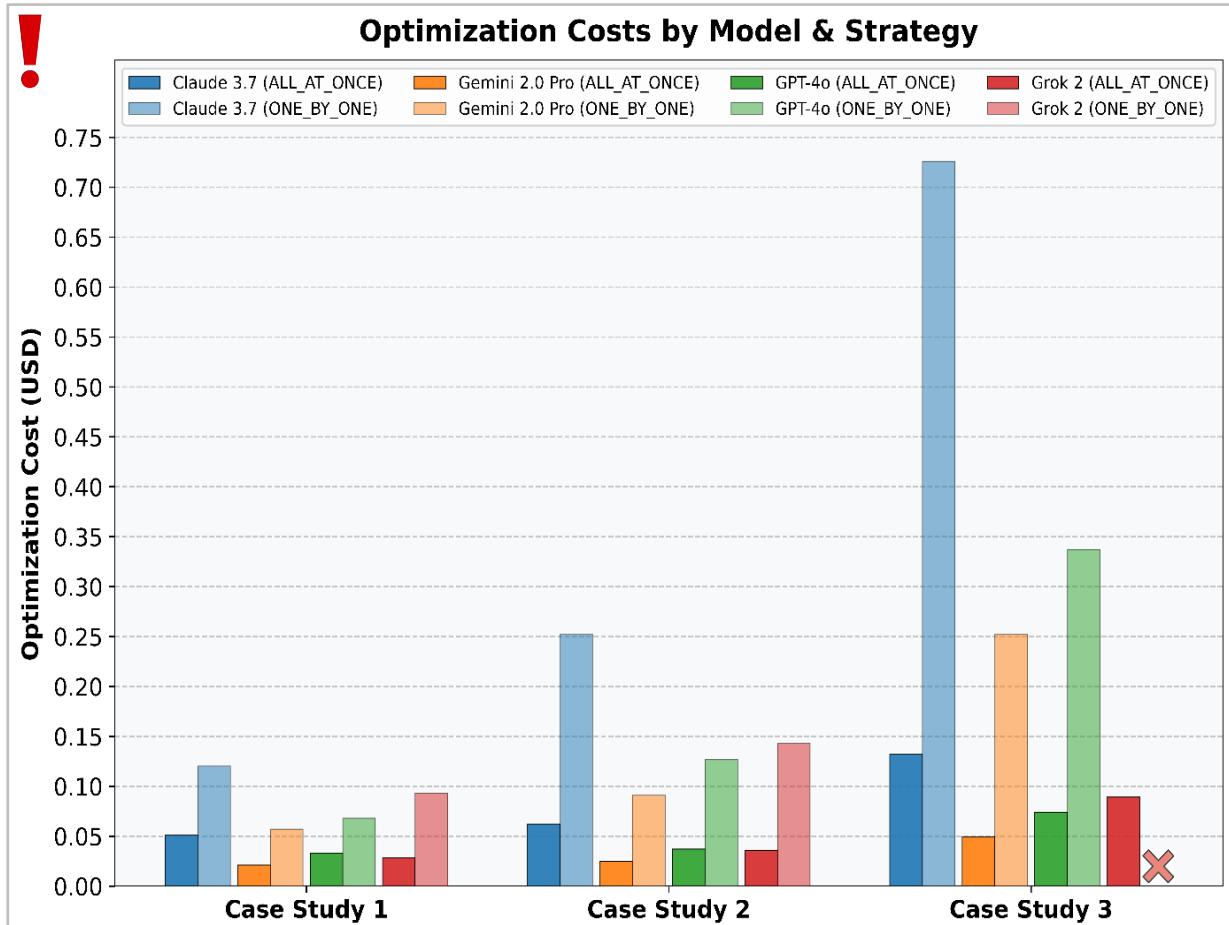


Abbildung 52: Vergleich der Optimierungskosten verschiedener LLMs und Strategien über alle drei Fallstudien hinweg

Abbildung 51 und Abbildung 52 verdeutlichen, dass die verschiedenen Anbieter unterschiedliche Tokenizer verwenden und teilweise trotz identischer Preisschemata erhebliche Unterschiede im Tokenverbrauch sowie den daraus resultierenden Optimierungskosten aufweisen. Dies wird insbesondere durch den Vergleich von gpt-4o-2024-08-06 mit grok-2-1212 ersichtlich: Obwohl gpt-4o auf den ersten Blick (in Bezug auf eine Million Tokens) teurer als grok-2 wirkt, ist es aufgrund der effizienteren Tokenization günstiger. Damit kann unterstrichen werden, dass neben den „offensichtlichen“ Modellkosten auch die Effizienz der Tokenverarbeitung eine zentrale Rolle bei der Wahl des geeigneten Modells spielt.

Das durchschnittlich kostenintensivste Modell ist claude-3-7-sonnet-20250219, gefolgt von grok-2-1212, gpt-4o-2024-08-06 und gemini-2.0-pro-exp-02-05.

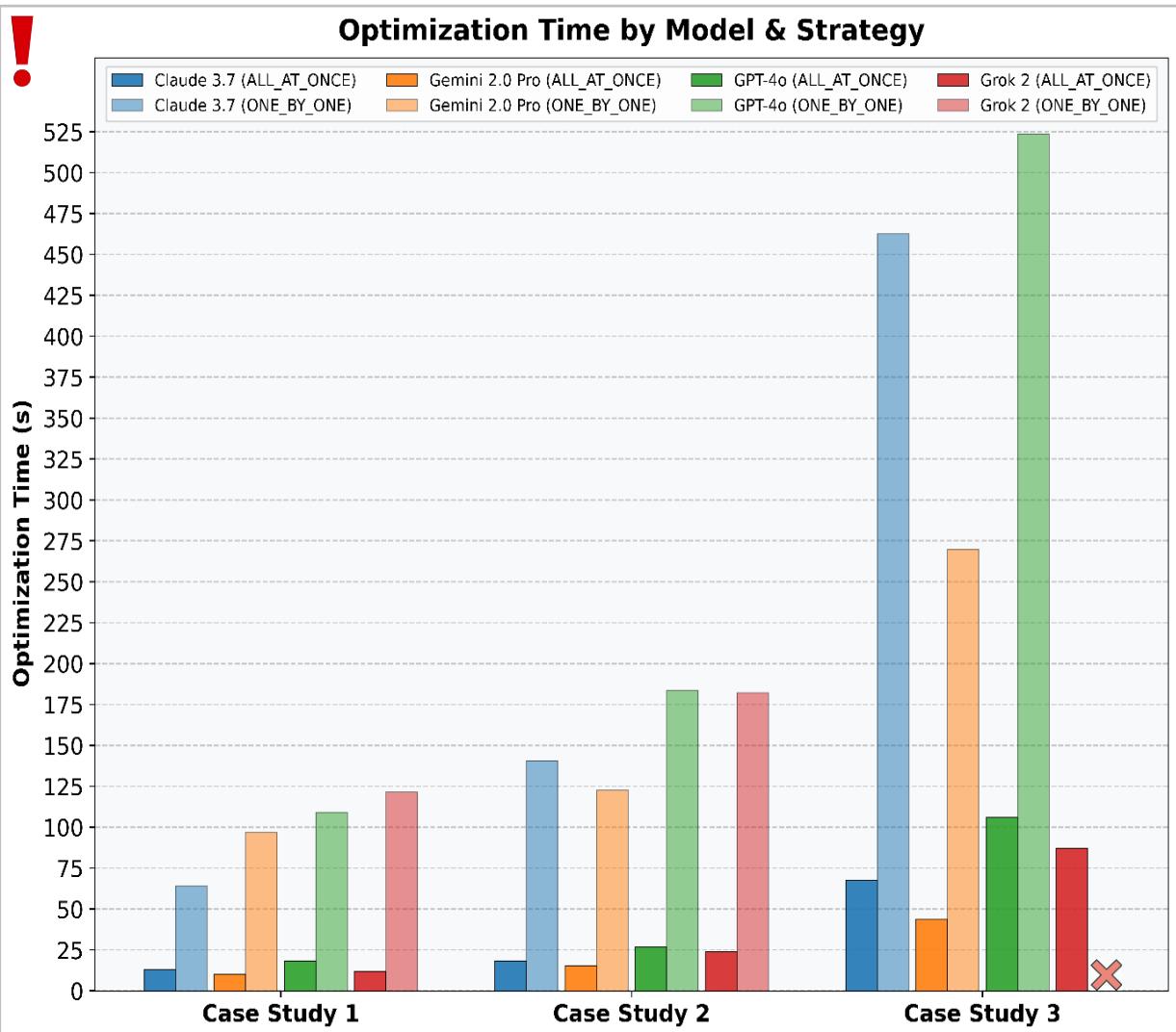


Abbildung 53: Vergleich der Optimierungsdauer verschiedener LLMs und Strategien über alle drei Fallstudien hinweg

Wie bereits erwartet, beeinflusst der Optimierungsansatz nicht nur die Kosten, sondern auch die Dauer des Optimierungsprozesses erheblich. Jedes Modell nimmt mit dem „One-by-One“-Ansatz in jeder Fallstudie sowohl mehr Kosten als auch Zeit in Anspruch.

Das durchschnittlich zeitintensivste Modell ist gpt-4o-2024-08-06, gefolgt von grok-2-1212, claude-3-7-sonnet-20250219 und gemini-2.0-pro-exp-02-05.

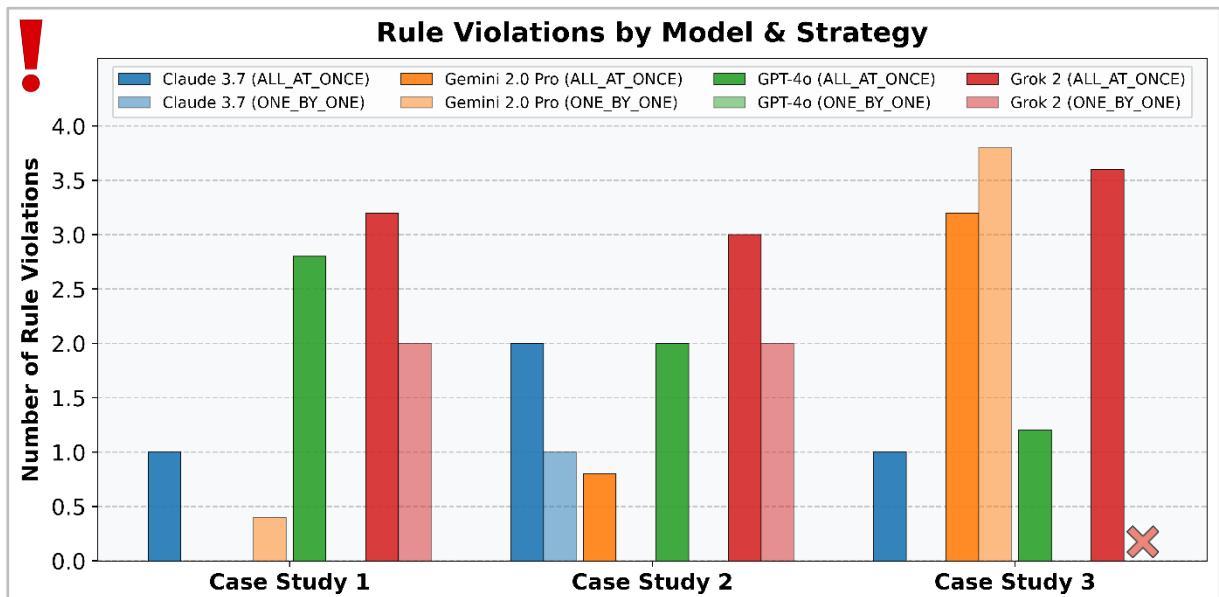


Abbildung 54: Vergleich der Regelverstöße verschiedener LLMs und Strategien über alle drei Fallstudien hinweg

Die vermutlich aussagekräftigste Metrik, zumindest im Rahmen dieser Arbeit, ist die Anzahl an Regelverstößen nach der Optimierung. Wie später bei der Ermittlung des optimalen Modells mithilfe einer Nutzwertanalyse ersichtlich wird, erhält dieser Parameter mit 60% eine hohe Gewichtung und spielt daher eine fundamentale Rolle. Denn erst, wenn die Optimierung qualitativ hochwertigen Code erzeugt, sollte der Fokus auf weitere Aspekte wie Geschwindigkeit und Kosten gelegt werden.

Abbildung 54 zeigt, dass alle Modelle – mit Ausnahme von gemini-2.0-pro-exp-02-05 – in Kombination mit dem „One-by-One“-Ansatz bessere Ergebnisse erzielen als mit dem „All-at-Once“-Ansatz. Dies legt nahe, dass eine sequenzielle Verarbeitung im Vergleich zu einer einmaligen Vorgehensweise tendenziell zuverlässigere und regelkonformere Ergebnisse liefert. Besonders die Ergebnisse von gpt-4o-2024-08-06 in Kombination mit dem „One-by-One“-Ansatz stechen heraus, da in keiner der Fallstudien ein Regelverstoß festgestellt werden konnte. Dazu aber später mehr (siehe Abbildung 58).

Da die Zuverlässigkeit von grok-2-1212 in der dritten Fallstudie nicht gewährleistet ist, werden in den folgenden Abbildungen nur claude-3-7-sonnet-20250219, gemini-2-0-pro-exp-02-05 und gpt-4o-2024-08-06 berücksichtigt. Würde grok-2 nur in einer einzelnen Fallstudie ausgeschlossen werden, so würde dies die Interpretierbarkeit der Ergebnisse beeinträchtigen.

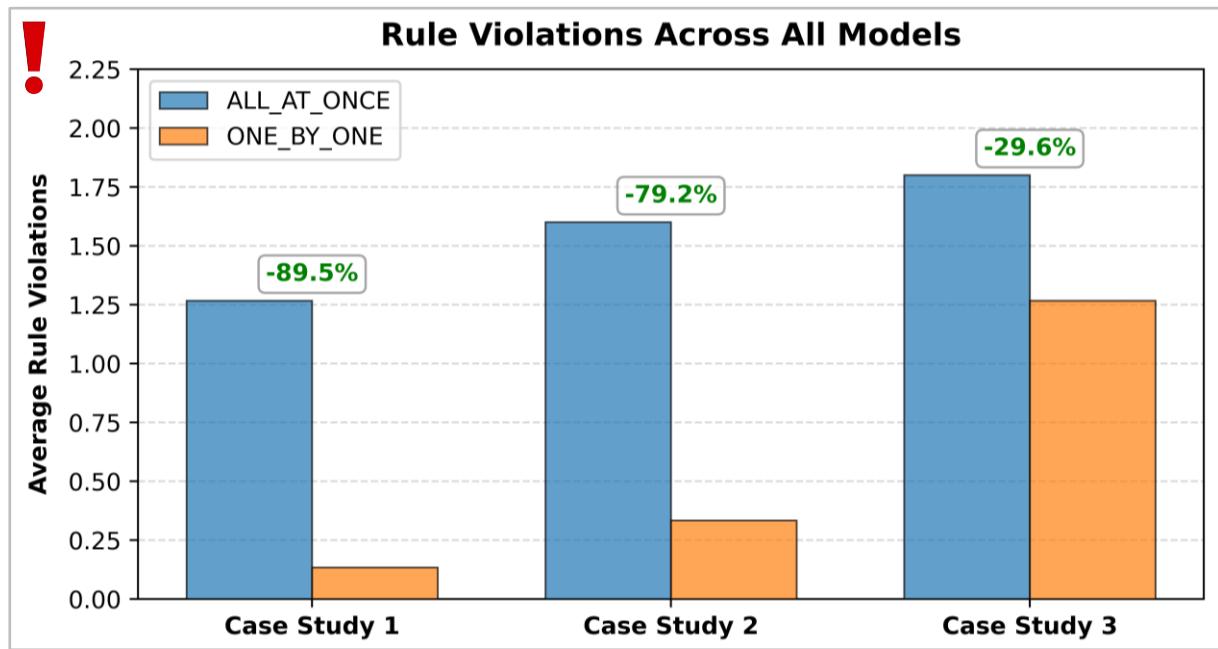


Abbildung 55: Durchschnittliche Anzahl an Regelverstößen über alle Ergebnisse der Modelle claude-3.7-sonnet-20250219, gemini-2.0-pro-exp-02-05 und gpt-4o-2024-08 hinweg

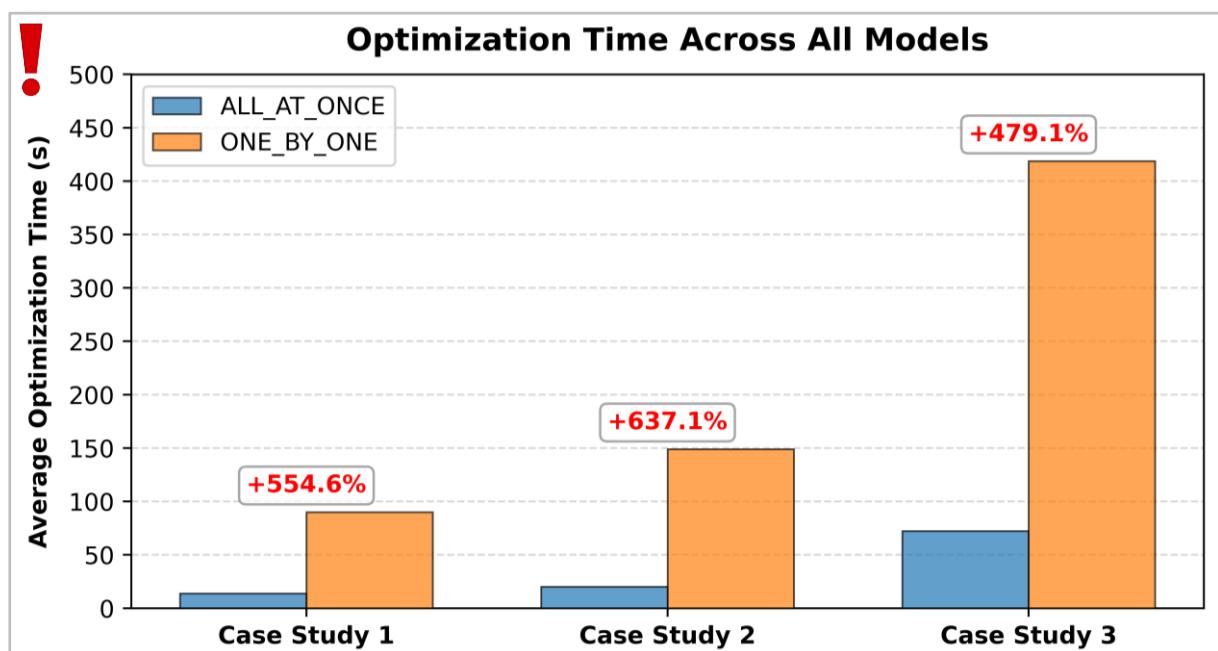


Abbildung 56: Durchschnittliche Optimierungsdauer über alle Ergebnisse der Modelle claude-3.7-sonnet-20250219, gemini-2.0-pro-exp-02-05 und gpt-4o-2024-08 hinweg

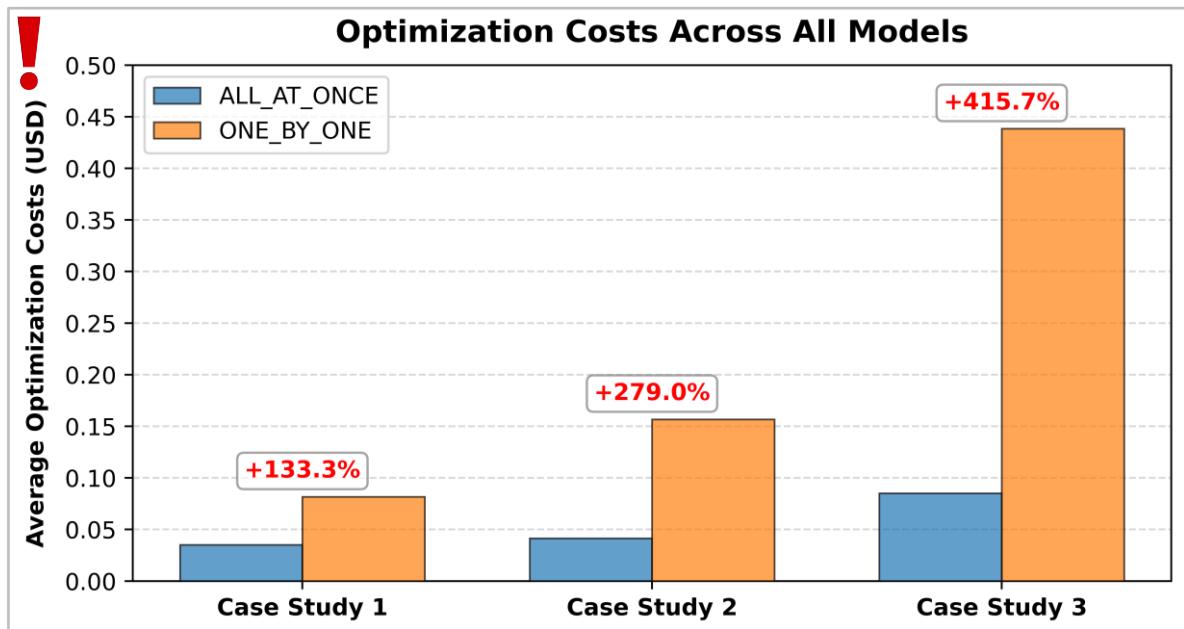


Abbildung 57: Durchschnittliche Optimierungskosten über alle Ergebnisse der Modelle Claude-3.7-Sonnet-20250219, Gemini-2.0-Pro-Exp-02-05 und GPT-4o-2024-08 hinweg

Zusammengefasst lässt sich der Vergleich zwischen den beiden Optimierungsansätzen aus Abbildung 55, Abbildung 56 und Abbildung 57 tabellarisch wie folgt darstellen:

Tabelle 18: Durchschnittliche Anzahl an Regelverstößen, Optimierungsdauer, Optimierungskosten über alle Ergebnisse der Modelle Claude-3.7-Sonnet-20250219, Gemini-2.0-Pro-Exp-02-05 und GPT-4o-2024-08 hinweg

		Average Rule Violations	Average Optimization Time [s]	Average Optimization Costs [USD]
Case Study 1	All-at-Once	1.27	13.73	0.035
	One-by-One	0.13	89.90	0.082
Case Study 2	All-at-Once	1.60	20.20	0.041
	One-by-One	0.33	148.90	0.157
Case Study 3	All-at-Once	1.80	72.30	0.085
	One-by-One	1.27	418.67	0.438

Tabelle 18 zeigt, dass der „All-at-Once“-Ansatz in sämtlichen Fallstudien schneller und kostengünstiger ist, jedoch mit mehr Regelverstößen einhergeht. Konträr dazu, reduziert der „One-by-One“-Ansatz die Anzahl an Regelverstößen, erfordert aber signifikant mehr Zeit und höhere Kosten. Welcher Ansatz verwendet werden sollte, lässt sich neutral mithilfe einer Nutzwertanalyse bestimmen. Dabei wird jedes Kriterium mit einer Gewichtung versehen, die dessen relative Bedeutung widerspiegelt. Danach wird das Produkt aus Bewertung bzw. Gewichtung des jeweiligen Kriteriums gebildet und aufaddiert. Dieser Prozess wird für jedes

Szenario durchgeführt und letztlich miteinander verglichen, wobei das Szenario mit dem höchsten Wert die bestmögliche Wahl darstellt. Für jedes Kriterium (X) wurde die Normalisierung mit der folgenden Formel berechnet, wobei ein Wert von 100 am besten und ein Wert von 0 am schlechtesten ist:

$$X_{norm} = 100 * \left(\frac{X_{min}}{X} \right) \quad (34)$$

X_{norm} ... Normierter Wert (zwischen 0 und 100)

X ... Ursprünglicher Wert des jeweiligen Kriteriums

X_{min} ... Minimaler (bester) Wert des jeweiligen Kriteriums

Tabelle 19: Nutzwertanalyse der Optimierungsstrategien der ersten Fallstudie über alle Modelle hinweg

Nutzwertanalyse Fallstudie 1 (über alle Modelle hinweg)					
Kriterium	Gewichtung	All-at-Once		One-by-One	
		X_{norm}	Wert	X_{norm}	Wert
Average Rule Violations	60%	10,5	6,3	100,0	60,0
Average Optimization Time	15%	100,0	15,0	15,3	2,3
Average Optimization Costs	25%	100,0	25,0	42,7	10,7
SUMME			52.,		73,0

Tabelle 20: Nutzwertanalyse der Optimierungsstrategien der zweiten Fallstudie über alle Modelle hinweg

Nutzwertanalyse Fallstudie 2 (über alle Modelle hinweg)					
Kriterium	Gewichtung	All-at-Once		One-by-One	
		X_{norm}	Wert	X_{norm}	Wert
Average Rule Violations	60%	20,8	12,5	100,0	60,0
Average Optimization Time	15%	100,0	15,0	13,6	2,0
Average Optimization Costs	25%	100,0	25,0	26,4	6,6
SUMME			52,5		68,6

Tabelle 21: Nutzwertanalyse der Optimierungsstrategien der dritten Fallstudie über alle Modelle hinweg

Nutzwertanalyse Fallstudie 3 (über alle Modelle hinweg)					
Kriterium	Gewichtung	All-at-Once		One-by-One	
		X_{norm}	Wert	X_{norm}	Wert
Average Rule Violations	60%	70,4	42,2	100,0	60,0
Average Optimization Time	15%	100,0	15,0	17,3	2,6
Average Optimization Costs	25%	100,0	25,0	19,4	4,8
SUMME			82,2		67,4

Die Ergebnisse zeigen, dass unabhängig vom verwendeten Modell der „One-by-One“-Ansatz für unerfahrene bzw. fortgeschrittene und der „All-at-Once“-Ansatz für professionelle Softwareentwickler:innen vorteilhaft ist. Wichtig ist dabei jedoch anzumerken, dass dies nur

allgemein über alle verwendeten Modelle hinweg gilt. Eine bedarfsgerechte Komprimierung an das hohe Erfahrungsniveau oder die Wahl eines spezifischen Modells könnte den Einsatz des „One-by-One“-Ansatz auch für professionelle Softwareentwickler:innen attraktiv gestalten.

Nun soll der Fokus darauf gerichtet werden, welches Modell in Kombination mit welcher Strategie besonders empfehlenswert ist und, ob der Einsatz des „One-by-One“-Ansatzes auch für professionelle Entwickler sinnvoll sein kann:

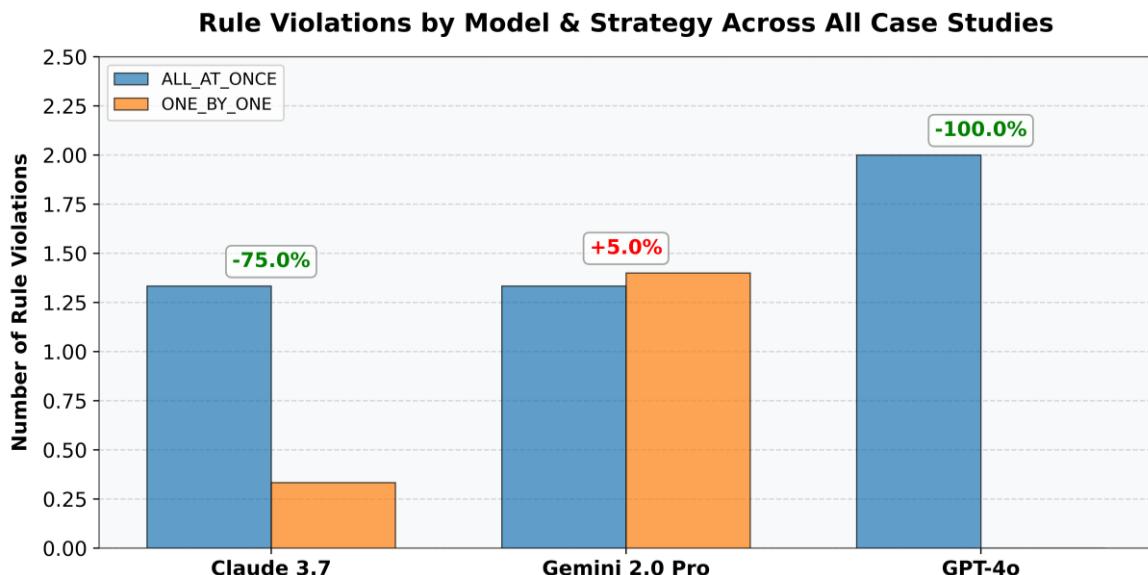


Abbildung 58: Durchschnittliche Anzahl an Regelverstößen über alle Fallstudien hinweg

Fundamental für die erfolgreiche Anwendung der Clean-Code-Prinzipien sind die nach der Optimierung verbliebenen Regelverstöße. Wie in Abbildung 58 ersichtlich, hat gpt-4o-2024-08-06 von OpenAI mit dem „One-by-One“-Ansatz in keiner der durchgeföhrten Messungen einen Regelverstoß verzeichnet. Daher wird diese Kombination als Standardmethode für OptiPy festgelegt. Es zeigt sich zudem, dass gemini-2.0-pro-exp-02-05 die Mittelwerte massiv verfälscht hat, weswegen eine weitere Nutzwertanalyse für das Modell (gpt-4o-2024-08-06) mit den geringsten Regelverstößen notwendig ist.

Tabelle 22: Durchschnittliche Anzahl an Regelverstößen, Optimierungsdauer, Optimierungskosten des gpt-4o-2024-08-06 Modells

		Average Rule Violations	Average Optimization Time [s]	Average Optimization Costs [USD]
Case Study 1	All-at-Once	2.80	18.40	0.033
	One-by-One	0.00	109.10	0.068
Case Study 2	All-at-Once	2.00	26.80	0.037
	One-by-One	0.00	183.50	0.127
Case Study 3	All-at-Once	1.20	105.80	0.074
	One-by-One	0.00	532.60	0.337

Tabelle 23: Nutzwertanalyse der Optimierungsstrategien der ersten Fallstudie für gpt-4o-2024-08-06

Nutzwertanalyse Fallstudie 1 (gpt-4o-2024-08-06)					
Kriterium	Gewichtung	All-at-Once		One-by-One	
		X _{norm}	Wert	X _{norm}	Wert
Average Rule Violations	60%	0,0	0,0	100,0	60,0
Average Optimization Time	15%	100,0	15,0	16,9	2,5
Average Optimization Costs	25%	100,0	25,0	49,0	12,2
SUMME			40,0		74,8

Tabelle 24: Nutzwertanalyse der Optimierungsstrategien der zweiten Fallstudie für gpt-4o-2024-08-06

Nutzwertanalyse Fallstudie 2 (gpt-4o-2024-08-06)					
Kriterium	Gewichtung	All-at-Once		One-by-One	
		X _{norm}	Wert	X _{norm}	Wert
Average Rule Violations	60%	0,0	0,0	100,0	60,0
Average Optimization Time	15%	100,0	15,0	14,6	2,2
Average Optimization Costs	25%	100,0	25,0	29,1	7,3
SUMME			40,0		69,5

Tabelle 25: Nutzwertanalyse der Optimierungsstrategien der dritten Fallstudie für gpt-4o-2024-08-06

Nutzwertanalyse Fallstudie 3 (gpt-4o-2024-08-06)					
Kriterium	Gewichtung	All-at-Once		One-by-One	
		X _{norm}	Wert	X _{norm}	Wert
Average Rule Violations	60%	0,0	0,0	100,0	60,0
Average Optimization Time	15%	100,0	15,0	19,9	3,0
Average Optimization Costs	25%	100,0	25,0	22,0	5,5
SUMME			40,0		68,5

Es zeigt sich somit, dass der Einsatz des „One-by-One“-Ansatzes je nach Modell auch für professionelle Entwickler sinnvoll sein kann. Deswegen kann untermauert werden, dass der Einsatz des gpt-4o-2024-08-06 Modells von OpenAI in Kombination mit dem „One-by-One“-Ansatz nicht nur für unerfahrene oder für fortgeschrittene, sondern auch für professionelle Softwareentwickler:innen empfehlenswert ist. Jedoch sollte der Clean-Code-Leitfaden, wie bereits erwähnt, bedarfsgerecht komprimiert und an das Erfahrungsniveau der Entwickler:innen angepasst werden, um sowohl die Genauigkeit zu verbessern als auch die Kosten und Dauer des Optimierungsprozess bestmöglich zu minimieren. Wichtig ist dabei zu berücksichtigen, dass die Ergebnisse und Handlungsempfehlungen dieser Arbeit auf drei Fallstudien beruhen. Obwohl dies bereits ein vielversprechendes Indiz dafür ist, dass LLMs eine zielgerichtete und automatisierte Anwendung von Clean-Code-Leitfäden ermöglichen, sollte vor der Einführung eines neuen Leitfadens oder der Anwendung in einem neuen Kontext ein neuer Benchmark durchgeführt werden.

Hinterfragung der Wirtschaftlichkeit:

Um sinnvolle Referenzwerte für die Berechnung sicherzustellen, wird sich an einer Fallstudie von Google orientiert. Dabei wurden über zwei Jahre 12 Interviews durchgeführt, 44 Teilnehmer befragt und 9 Millionen Änderungen aus Review-Protokollen analysiert. Es wurde festgestellt, dass Code-Reviews primär der Lesbarkeit bzw. Wartbarkeit und nicht der Fehlerfindung dienen. Des Weiteren beträgt der durchschnittliche Aufwand drei Stunden pro Woche, wobei ein Reviewer als ausreichend angesehen wird. Der Median an Dateien, die ein Mitarbeiter nach 18 Monaten überprüft hat, beträgt in etwa 400. Dies entspricht 266,67 Dateien pro Jahr, 22,22 Dateien pro Monat, 5,13 Dateien pro Woche oder ~1 Datei pro Arbeitstag. Damit ergibt sich der durchschnittliche Aufwand einer Datei von 35,09 Minuten. [127]

Der typische Code-Review läuft dabei wie folgt ab [127]:

- **Creating:** Der Autor reicht Code-Änderungen zur Überprüfung ein.
- **Previewing:** Der Reviewer begutachtet Code-Änderungen und bereitet Feedback vor.
- **Commenting:** Der Reviewer kommentiert Anmerkungen direkt im Code.
- **Addressing Feedback:** Der Autor passt den Code entsprechend den Kommentaren an.
- **Approving:** Wenn alle Probleme behoben sind, bestätigt der Reviewer die Änderungen.

Die durchschnittlichen Arbeitsstunden pro Jahr werden auf Basis einer 38,5h/Woche wie folgt berechnet:

Gesamtes Jahr	52 $\frac{\text{Wochen}}{\text{Jahr}}$	260 $\frac{\text{Tage}}{\text{Jahr}}$	2002 $\frac{\text{Stunden}}{\text{Jahr}}$
– Urlaub	-5 $\frac{\text{Wochen}}{\text{Jahr}}$	-25 $\frac{\text{Tage}}{\text{Jahr}}$	-192,5 $\frac{\text{Stunden}}{\text{Jahr}}$
– Krankenstand	-3 $\frac{\text{Wochen}}{\text{Jahr}}$	-15 $\frac{\text{Tage}}{\text{Jahr}}$	-115,5 $\frac{\text{Stunden}}{\text{Jahr}}$
– Feiertage	-2 $\frac{\text{Wochen}}{\text{Jahr}}$	-10 $\frac{\text{Tage}}{\text{Jahr}}$	-77 $\frac{\text{Stunden}}{\text{Jahr}}$
– Sonstige Verhinderungen (z.B.: Schulungen)	-1 $\frac{\text{Woche}}{\text{Jahr}}$	-5 $\frac{\text{Tage}}{\text{Jahr}}$	-38,5 $\frac{\text{Stunden}}{\text{Jahr}}$
= Nutzbare Arbeitszeit	= 41 $\frac{\text{Wochen}}{\text{Jahr}}$	= 205 $\frac{\text{Tage}}{\text{Jahr}}$	= 1578,5 $\frac{\text{Stunden}}{\text{Jahr}}$

Das Bruttostundenentgelt ergibt sich folgendermaßen:

$$\begin{aligned}
 & \text{Arbeitnehmerbrutto} && 4500 \frac{\text{€}}{\text{Monat}} \\
 & \text{Bruttostundenentgelt} && \frac{4500 \frac{\text{€}}{\text{Monat}} * 12 \frac{\text{Monate}}{\text{Jahr}}}{1578,5 \frac{\text{Stunden}}{\text{Jahr}}} = 34,21 \frac{\text{€}}{\text{Stunde}}
 \end{aligned}$$

Die Nebenkosten werden mithilfe des Lohnnebenkostenrechners 2025 vom Bundesministerium für Finanzen (<https://bmf.lohn365.at/bmf/Lohnkostenrechner>) auf Basis der Anwesenheitszeit berechnet:

Anwesenheits-Entgelt	100,00 %
+ Nichtanwesenheits-Entgelt	+ 26,71 %
= Laufende Bezüge	= 126,71 %
+ Sonderzahlung	+ 21,12 %
= Direkte Arbeitskosten	= 147,83 %
+ Summe Sozialabgaben Ifd	+ 37,47 %
+ Summe Sozialabgaben SZ	+ 6,14 %
+ Sonstige Kosten	+ 8,74 %
= Nebenkosten	= 99,84 %

Schließlich können die Arbeitgeberbruttokosten pro Minute berechnet werden:

Bruttostundenentgelt	34,21 $\frac{\text{€}}{\text{Stunde}}$
+ Nebenkosten	+ 34,21 $\frac{\text{€}}{\text{Stunde}} * 9,84 \%$
= Arbeitgeberbruttokosten	= 66,81 $\frac{\text{€}}{\text{Stunde}}$ = 1,11 $\frac{\text{€}}{\text{Minute}}$

Die durchschnittlichen Kosten pro Zeile werden auf Basis des empfohlenen gpt-4o-2024-08-06 Modells in Kombination mit dem „One-by-One“-Ansatz berechnet. Dabei zeigt sich, dass desto höher die Anzahl an Zeilen, desto geringer die Optimierungskosten pro Zeile:

Tabelle 26: Berechnung der durchschnittlichen Optimierungskosten pro Zeile für das gpt-4o-2024-08-06 Modells in Kombination mit dem „One-by-One“-Ansatz

LOC	Average Optimization Costs [USD]	Average Optimization Costs per Line [USD/Line]
Case Study 1	32	0,068
Case Study 2	87	0,127
Case Study 3	331	0,337

Da die Fallstudie von Google [127] keine genaue Angabe zur Anzahl der in den genannten 35,09 Minuten überprüften Zeilen macht, wird angenommen, dass es sich ungefähr um den

Umfang der dritten Fallstudie handelt. Damit ergibt sich ein Wert von 9.43 Zeilen pro Minute, welcher durchaus realistisch erscheint. In USD umgewandelt ergibt sich aus der dritten Fallstudie ein Wert von 0,0009433€/Zeile (Stand: 26.03.2025).

Die Infrastrukturkosten belaufen sich bei einem Github-Unternehmensaccount auf 21 USD pro Entwickler:in pro Monat [128]. Dies entspricht 1,2292683 USD (bei 205 Arbeitstagen pro Jahr) bzw. 1,1390400€/Arbeitstag (Stand: 26.03.2025) pro Entwickler:in pro Arbeitstag.

Die Instandhaltungskosten werden unternehmensweit mit einem Aufwand von einer Stunde pro Arbeitswoche angenommen. Heruntergebrochen ergeben sich somit 12 Minuten pro Arbeitstag. Des Weiteren wird angenommen, dass die Instandhaltung von einem Entwickler übernommen wird. Die geringen Kosten sind auf die aufwandsarme Implementierung von OptiPy mithilfe der bereitgestellten CI/CD-Pipeline zurückzuführen.

Die Kosten, die sich ein Unternehmen durch den Einsatz von OptiPy pro Entwickler:in einspart, werden folgendermaßen berechnet. Dabei bezieht sich die Gleichung auf die Ersparnis pro Entwickler:in:

$$G_{Entwickler:in} = [t * p * c] - \left[(l * o) + i + \left(\frac{m * x * c}{n} \right) \right] \quad (35)$$

G_{Entwickler:in} ... Eingesparte Kosten pro Entwickler:in pro Arbeitstag durch OptiPy [€/Tag]

t ... Zeitaufwand für Code-Reviews pro Entwickler:in pro Arbeitstag [min/Tag]

p ... Prozentualer Zeitgewinn eines Code-Reviewers durch OptiPy pro Entwickler:in [%]

c ... Arbeitgeberbruttokosten pro Entwickler:in pro Minute [€/min]

l ... Zeilenanzahl pro Entwickler:in pro Code-Review pro Arbeitstag [Zeilen/Tag]

o ... Durchschnittliche Kosten pro überprüfter Codezeile durch OptiPy [€/Zeile]

i ... Infrastrukturkosten pro Entwickler:in pro Arbeitstag [€/Tag]

m ... Zeitaufwand für Instandhaltung von OptiPy (für gesamte Userbase) pro Arbeitstag [min/Tag]

x ... Anzahl an Entwickler:innen, die für die Instandhaltung verantwortlich sind

n ... Anzahl an Entwickler:innen im Unternehmen

Für den finalen Wirtschaftlichkeitsvergleich werden, wie bereits erwähnt, folgende statische Parameter verwendet:

- **t = 35,09 $\frac{\text{min}}{\text{Tag}}$**
- **c = 1,11 $\frac{\text{€}}{\text{min}}$** (Auf Basis eines Arbeitnehmerbruttos von 4500 $\frac{\text{€}}{\text{Monat}}$)
- **l = 331 $\frac{\text{Zeilen}}{\text{Tag}}$**
- **o = 0,0009433 $\frac{\text{€}}{\text{Zeile}}$**
- **i = 1,1390400 $\frac{\text{€}}{\text{Tag}}$** (Auf Basis einer monatlichen Mitgliedschaft von 21 $\frac{\text{USD}}{\text{Monat}}$)

- $m = 15,00 \frac{\text{min}}{\text{Tag}}$
- $x = 1$

Letztlich werden diese Variablen in Gleichung (35) eingesetzt und der prozentuale Zeitgewinn eines Code-Reviewers durch die Nutzung von OptiPy p bzw. die Anzahl an Entwickler:innen im Unternehmen n variiert. Dabei werden folgende drei Unternehmensgrößen berücksichtigt [129], wobei die Anzahl an Entwickler:innen ohne Berücksichtigung weiterer Mitarbeiter:innen aus anderen Abteilungen angegeben wird:

- Kleinunternehmen (10 Entwickler)
- Mittlere Unternehmen (100 Entwickler)
- Großunternehmen (1000 Entwickler)

Die Umrechnung von $G_{\text{Entwickler:in}}$ basiert auf der realen Arbeitszeit, das heißt 1 Jahr entspricht 205 Arbeitstagen. Die eingesparten Kosten des gesamten Unternehmens G_{Gesamt} bilden sich aus dem Produkt von $G_{\text{Entwickler:in}}$ und n .

Tabelle 27: Wirtschaftlichkeitsvergleich der eingesparten Arbeitgeberbruttokosten pro Entwickler:in pro Jahr durch OptiPy in einem Kleinunternehmen mit 10 Entwickler:innen

Kleinunternehmen				
	n	p	$G_{\text{Entwickler:in}}$	G_{Gesamt}
Worst Case	10	0%	-570,44€/Jahr	-5 704,40€/Jahr
Base Case	10	33%	+2 064,52€/Jahr	+20 645,20€/Jahr
Best Case	10	66%	+4 699,48€/Jahr	+46 994,80€/Jahr

Tabelle 28: Wirtschaftlichkeitsvergleich der eingesparten Arbeitgeberbruttokosten pro Entwickler:in pro Jahr durch OptiPy in einem mittleren Unternehmen mit 100 Entwickler:innen

Mittlere Unternehmen				
	n	p	$G_{\text{Entwickler:in}}$	G_{Gesamt}
Worst Case	100	0%	-324,69€/Jahr	-32 469,00€/Jahr
Base Case	100	33%	+2 310,27€/Jahr	+231 027,00€/Jahr
Best Case	100	66%	+4 945,23€/Jahr	+494 523,00€/Jahr

Tabelle 29: Wirtschaftlichkeitsvergleich der eingesparten Arbeitgeberbruttokosten pro Entwickler:in pro Jahr durch OptiPy in einem Großunternehmen mit 1000 Entwickler:innen

Großunternehmen				
	n	p	$G_{\text{Entwickler:in}}$	G_{Gesamt}
Worst Case	1000	0%	-300,11€/Jahr	-300 110,00€/Jahr
Base Case	1000	33%	+2 334,85€/Jahr	+2 334 850,00€/Jahr
Best Case	1000	66%	+4 969,81€/Jahr	+4 969 810,00€/Jahr

Werden die Zusammenhänge grafisch dargestellt, ergibt sich folgende Abbildung:

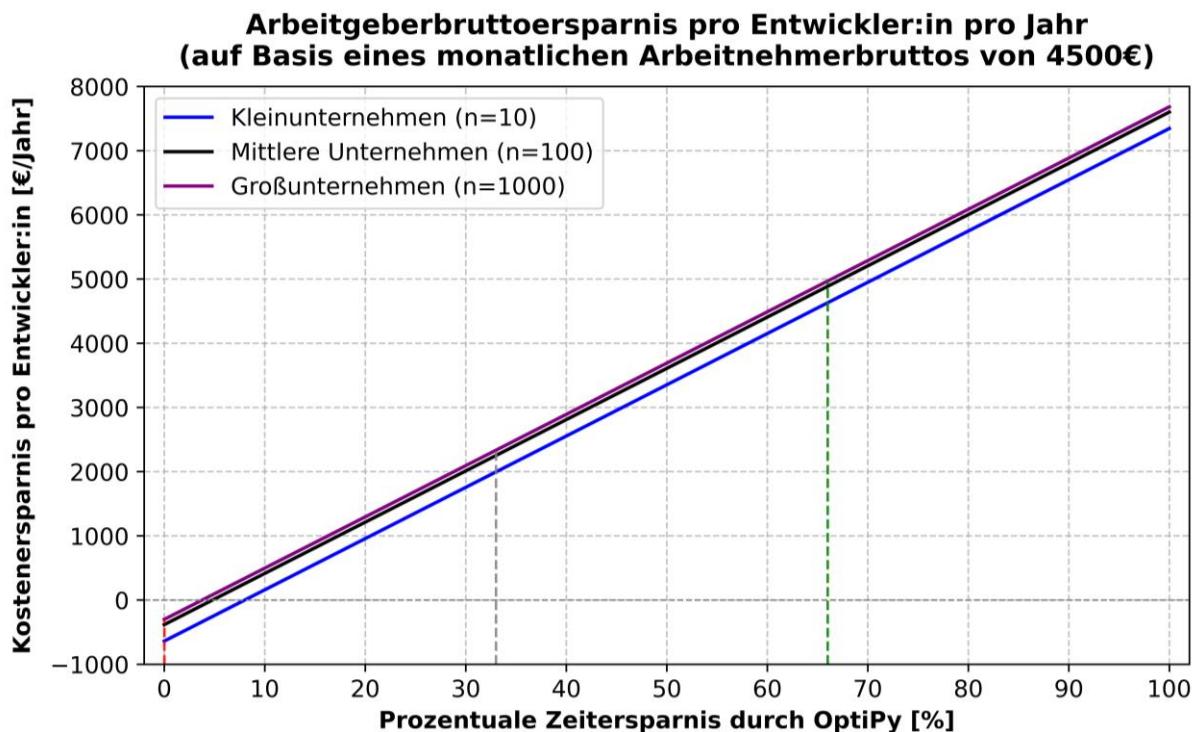


Abbildung 59: Arbeitgeberbruttoersparnis pro Entwickler:in pro Jahr in Abhängigkeit der prozentualen Zeitersparnis durch OptiPy auf Basis eines monatlichen Arbeitnehmerbruttos von 4500€

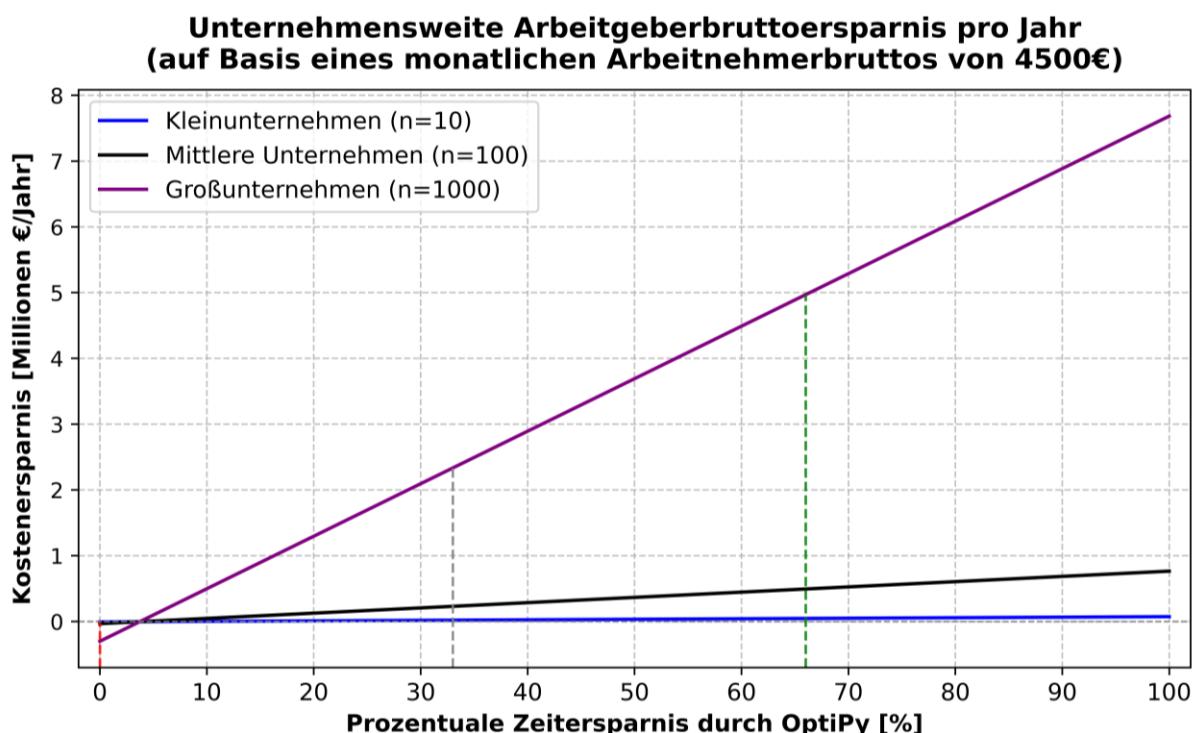


Abbildung 60: Arbeitgeberbruttoersparnis pro Jahr in Abhängigkeit der prozentualen Zeitersparnis durch OptiPy auf Basis eines monatlichen Arbeitnehmerbruttos von 4500€

Hinweis: Bei dem angenommenen monatlichen Bruttogehalt handelt es sich um einen durchschnittlichen Orientierungswert. In Großunternehmen ist das Gehalt typischerweise höher als in kleinen oder mittleren Betrieben – beispielweise durch Boni in Form eines 15. Gehalts oder durch überkollektivvertragliche Zahlungen. Aus Gründen der besseren Nachvollziehbarkeit wurde jedoch für alle Unternehmensgrößen dasselbe Gehalt angenommen.

Der Break-Even-Point befindet sich bei Kleinunternehmen mit 10 Entwickler:innen bei einer prozentualen Zeitersparnis von 8,00%, bei mittleren Unternehmen mit 100 Entwickler:innen bei 4,15% und bei Großunternehmen mit 1000 Entwickler:innen bei 3,77%. Unter diesem Schwellenwert entstehen dem Unternehmen Mehrkosten – ab diesem Schwellenwert beginnen Einsparungen. Der geringe Break-Even-Point verdeutlicht, dass bereits marginale Zeitersparnisse im Kontext des Code-Reviews ausreichen, um die Wirtschaftlichkeit von OptiPy zu rechtfertigen. Wie bereits erwartet beeinflussen die Instandhaltungskosten den Break-Even-Point deutlich: Desto mehr Entwickler:innen im Unternehmen beschäftigt sind, desto höher ist die Kosteneinsparung pro Mitarbeiter.

Dabei gilt natürlich folgender Aspekt zu beachten: Üblicherweise werden die Entwicklungskosten berücksichtigt, wodurch sich der Nulldurchgang erheblich verzögert. Da das Tool bereits entwickelt ist, werden diese Kosten im Rahmen dieser Arbeit nicht miteinbezogen.

Summa Summarum kann festgestellt werden, dass eine automatisierte Anwendung des entwickelten Clean-Code Leitfadens durch ein Large Language Model wirtschaftlich durchaus sinnvoll ist.

4.2 Diskussion

Hinterfragung der Zielgruppe von OptiPy

Insbesondere für Anfänger oder Teams, die keine Kapazitäten haben, um sich mit komplexen Qualitätsmodellen, wie beispielweise dem Arc Q42 Modell auseinanderzusetzen und mit eigenen Verbesserungsmaßnahmen aufzukommen, empfiehlt sich der Einsatz des in dieser Arbeit entwickelten Tools. Neben der automatisierten Anwendung des Leitfadens und einer damit verbundenen Verbesserung der inneren Softwarequalität, konkret der Maintainability, der Performance Efficiency und der Reliability, von Python-Applikationen, wird eine Brücke zwischen unerfahrenen bzw. fortgeschrittenen Softwareentwickler:innen bzw. MINT-Absolventen und professionellen Softwareentwickler:innen geschlagen. Dadurch kann trotz unzureichender Praxis- bzw. Branchenerfahrung eine qualitativ hochwertige Software bereitgestellt werden. Für professionelle Softwareentwickler:innen ist eine bedarfsgerechte Komprimierung des Clean-Code-Leitfadens empfehlenswert.

Grenzen des OpenAI-Promptings

Bei ersten experimentellen Tests wurden Programme mit einer höheren Zeilenanzahl überprüft. Hier konnte festgestellt werden, dass die Dauer des Optimierungsprozesses maßgeblich von der Länge des zu optimierenden Python-Files abhängt. Hier sollte überprüft werden für welche Programmlänge eine sinnvolle Optimierung gewährleistet werden kann.

Risiko bei Ausfall oder Wartungsarbeiten von bzw. an LLMs:

Während Tests am 12.12.2024 um 01:00 war OpenAI aufgrund eines unvorhersehbaren Systemausfalls nicht erreichbar. Obwohl der Ausfall weniger als eine Stunde gedauert hat, wurde dadurch die Abhängigkeit von einzelnen Anbietern deutlich aufgezeigt. Hier bedarf es deshalb einer Evaluierung von anderen LLM-Anbietern, um zukünftig in ähnlichen Situationen einen zuverlässigen Alternativplan sicherzustellen.

Alternativer „semi-automatisierter“ Optimierungsprozess

Der in jedem Fall zutreffende Weg scheint eine Kombination aus automatisierter Optimierung mittels LLM-Prompting inkl. anschließender manueller Überprüfung zu sein. Beispielseweise vor einem Meilenstein, wie einer Präsentation vor oder der Lieferung an einen Kunden, wird zuerst ein Großteil der Fehler automatisiert korrigiert, bevor der Qualitätsprüfer die abschließende manuelle Überprüfung durchführt. Dadurch wird der Qualitätsprüfer entlastet und kann sich auf andere anspruchsvollere Aufgaben fokussieren. Damit können nicht nur personelle Ressourcen, sondern letztlich auch Kosten eingespart werden.

4.3 Weitere Vorgehensweise

1. Optionale Optimierung der Performance Efficiency durch Prompt-Erweiterung

Derzeit fokussiert sich das in dieser Arbeit entwickelte Tool auf Clean-Code-Prinzipien. Daher empfiehlt es sich, die Möglichkeit einer optionalen Optimierung der Performance des Python-Programmes zu implementieren. Dies könnte beispielsweise durch das Hinzufügen eines weiteren Iterationsschritts am Anfang des Prozesses mit Angabe des folgenden Prompts erfolgen:

- Maximize algorithmic efficiency in terms of both runtime and memory (Big-O notation).
- Utilize parallelization and vectorization where appropriate.

2. Integration eines Directory-basierten Optimierungsprozesses

Derzeit ist OptiPy darauf ausgelegt, einzelne Python-Dateien zu optimieren. Obwohl dies in den meisten Anwendungsfällen völlig ausreichend ist, kann es in großen Softwareprojekten jedoch ineffizient sein. Werden beispielweise die unter den verschiedenen Python-Dateien bestehenden Abhängigkeiten gelöscht, kann die Funktionalität des gesamten Projekts eingeschränkt werden. Daher sollte die Sinnhaftigkeit eines Directory-basierten Optimierungsprozesses hinterfragt werden.

3. Kontinuierliche Verbesserung des Leitfadens & des verwendeten LLMs

Der Clean-Code-Leitfaden sollte laufend erweitert werden, um die mit neuen Python-Versionen eingeführten Optimierungsmaßnahmen zu berücksichtigen. Darüber hinaus ist es empfehlenswert den Leitfaden nach einer ausreichenden Testphase mit weiteren Clean-Code-Prinzipien zu ergänzen. Darunter fällt zum Beispiel die Implementierung von Exceptions mit aussagkräftigen Fehlermeldungen. Des Weiteren sollten laufend Benchmarks durchgeführt werden, um neu veröffentlichte Modelle hinsichtlich ihrer Leistungsfähigkeit zu evaluieren.

4. Verwendung verschiedener Temperaturen & Modelle für spezifische Kapitel

Im Rahmen dieser Arbeit wurde für die jeweiligen Optimierungen stets dasselbe Modell mit einer Temperatur von Null verwendet. Es ist interessant zu analysieren, ob die dynamische Anpassung des Modells oder der Temperatur für jedes Clean-Code-Kapitel eventuell bessere Ergebnisse erzielt. Beispielsweise könnte das Refactoring durch claude-3-7-sonnet-20250219 in komplexeren Sachverhalten zu besseren Ergebnissen führen (siehe Abbildung 48). Insbesondere für Dokumentationszwecke, wie Docstrings, könnte eine minimal erhöhte Temperatur eventuell kreativere und variantenreichere Beschreibungen erzielen.

5. Erstellung einer Benutzeroberfläche mit Einbeziehung von Streams

Eine weitere mögliche Vorgehensweise besteht darin, eine Benutzeroberfläche unter Berücksichtigung von Streams zu erstellen, um den Softwareentwickler:innen eine interaktive und dynamische Umgebung zu bieten. Dabei sollte jedoch der Mehrwert gegenüber bereits verfügbaren Benutzeroberflächen, wie OpenAI Playground oder Anthropic Workbench, hinterfragt werden.

6. Fine-Tuning des LLMs zur Steigerung der Genauigkeit und Effizienz

Nachdem das entwickelte Tool ausreichend getestet und verwendet wurde, könnten die optimierten Python-Dateien verwendet werden, um damit ein Fine-Tuning durchzuführen. Von den in dieser Arbeit behandelten Modellen ist es lediglich möglich das gpt-4o-2024-08-06 Modell zu fine-tunen. Ob ein, Fine-Tuning tatsächlich sinnvoll ist, hängt davon ab, ob das Verhalten des Modells in Kombination mit dem „One-by-One“-Ansatz auch bei anderen Python-Dateien erfolgreich ist. Sollte es der Fall sein, dass die Regelverstöße in anderen Anwendungsfällen zunehmen, so könnte ein Fine-Tuning ratsam sein. Dabei sollten stets die erläuterten Informationen aus 2.7.4 Fine-tuning berücksichtigt werden.

7. Automatische Ersetzung fest einprogrammierter Variablen

Das Python-Modul dotenv [127] liest key-value pairs aus einer .env-Datei ein und setzt diese als Umgebungsvariablen. Es unterstützt die Entwicklung von Anwendungen, insbesondere jenen die den 12-Factor-Principles folgen. Bei den 12-Factor-Principles handelt es sich um eine Zusammenführung idealer Praktiken von Heroku, einer Cloud Application Platform, um skalierbare und wartbare Software-as-a-Service (SaaS) gewährleisten zu können. Im Rahmen dieser Arbeit wird diese Methodik aber nicht genauer erläutert, da dies den Rahmen sprengen würde. Eine nähere Erklärung kann unter [131] abgerufen werden.

Die .env-Datei könnte beispielweise wie folgt aussehen:

```
AUTH_KEY="SuperSecretKey"
```

Ist es der Fall, dass sich eine Variable über mehrere Zeilen erstreckt, kann dies auf folgende beide Arten definiert werden:

```
FOO="first line\nsecond line"
FOO="first line
second line"
```

Um nun die Variablen aus der .env-Datei in ein beliebiges Python-File laden zu können, ist die Verwendung von folgendem Code notwendig:

```
import os
from dotenv import load_dotenv

load_dotenv()
AUTH_KEY = os.getenv("AUTH_KEY")
```

Wichtig bei der Verwendung von `load_dotenv()` ist folgendes:

```
If both `dotenv_path` and `stream` are `None`, `find_dotenv()` is used to find
the .env file.
```

Das bedeutet, dass `find_dotenv()` automatisch nach einer .env-Datei sucht und `load_dotenv()` die darin enthaltenen Umgebungsvariablen erkennt und so innerhalb des Python-Files abrufbar macht.

Es kann jedoch der Fall sein, dass der Pfad der .env-Datei von dem Pfad des Arbeitsverzeichnisses abweicht, weshalb dies auch berücksichtigt werden muss:

```
import os
from dotenv import load_dotenv
from pathlib import Path

dotenv_path = Path("path/to/.env")
load_dotenv(dotenv_path=dotenv_path)
AUTH_KEY = os.getenv("AUTH_KEY")
```

Ist es jedoch der Fall, dass bereits eine Umgebungsvariable mit demselben Namen erstellt oder importiert wurde, so sollte die standardmäßige Einstellung der `load_dotenv` Funktion mithilfe von `override=True` überschrieben werden.

```
from dotenv import load_dotenv

load_dotenv(override=True)
```

Um nun aber das maximale Potential aus dem bequemen Import einer .env-Datei ausreizen zu können, ist die Einführung und Anpassung einer .gitignore-Datei notwendig (vorausgesetzt die Versionsverwaltung erfolgt via Git).

```
.env
```

Dadurch wird gewährleistet, dass sämtliche vertrauliche Variablen lediglich lokal und nicht remote gespeichert werden. Doch warum das Ganze? Die Praxis zeigt, dass täglich streng vertrauliche Daten, wie API-Keys, unabsichtlich von Entwicklern, beispielweise auf Github, veröffentlicht werden. Im schlimmsten Fall handelt es sich dabei auch noch um ein öffentliches Repository, welches, wie der Name schon offenbart, von jeglichen Personen einsehbar ist.

Insbesondere durch die Einführung von kostenpflichtigen Features bei beliebten 3rd-Party-Anbietern, wie zum Beispiel bei OpenAI, kann eine unabsichtliche Veröffentlichung erhebliche finanziellen Schäden verursachen. Sind solche API-Keys nämlich öffentlich zugänglich, so könnten böswillige Akteure fremde und vom Besitzer nicht autorisierte Dienste in Anspruch nehmen. Inzwischen gibt es auch sogenannte Scraper, welche automatisiert nach „geleakten“ (unabsichtlich veröffentlichten) API-Keys suchen. Aus genannten Gründen ist es insbesondere für Einsteiger oder Junior-Entwickler empfehlenswert früh mit dieser praxisnahen Sicherheitsvorkehrung in Kontakt zu treten.

Hinweis: Bei den meisten Anbietern, können auch spezifische IP-Adressen in eine sogenannte Whitelist eingetragen werden, um Unbefugten den Zugriff zu verwehren. Selbst, wenn der API-Key in falsche Hände gelangt, ist die Nutzung lediglich, nur von den in der Whitelist hinterlegten IP-Adressen erlaubt. Oftmals kann darüber hinaus auch ein maximales Nutzungslimit auf Tages- oder Monatsbasis eingestellt werden. Da diese Methoden nicht bei allen Anbietern verfügbar sind und leider in der Praxis nicht intensiv genug benutzt werden, ist die Nutzung von Umgebungsvariablen essenziell.

Literaturverzeichnis

- [1] P. Carbonnelle, "PYPL PopularitY of Programming Language", 2025. [Online]
Verfügbar: <https://pypl.github.io/PYPL.html> (Zugriff am 24. Februar 2025)
- [2] P. Jansen, "TIOBE Index", TIOBE Software BV, 2025. [Online]
Verfügbar: <https://www.tiobe.com/tiobe-index/> (Zugriff am 24. Februar 2025)
- [3] P. Jansen, "TIOBE Programming Community Index Definition", TIOBE Software BV, 2025. [Online]
Verfügbar: https://www.tiobe.com/tiobe-index/programminglanguages_definition/ (Zugriff am 24. Februar 2025)
- [4] Google Trends, "Interest Over Time: Python vs MATLAB", Google LLC., 2025. [Online]
Verfügbar:
https://trends.google.com/trends/explore?date=all&q=%2Fm%2F05z1,%2Fm%2F053_x&hl=en-GB (Zugriff am 24. Februar 2025)
- [5] Stack Overflow Tag Trends, "Python vs MATLAB", Stack Exchange Inc., 2025. [Online]
Verfügbar: <https://trends.stackoverflow.co/?tags=python,matlab> (Zugriff am 24. Februar 2025)
- [6] OpenAI, "Introducing ChatGPT", OpenAI Inc., 2022. [Online]
Verfügbar: <https://openai.com/index/chatgpt/> (Zugriff am 24. Februar 2025)
- [7] W. Harding, " AI Copilot Code Quality: Evaluating 2024's Increased Defect Rate via Code Quality Metrics", GitClear by alloy.dev, 2025. [Online]
Verfügbar: https://www.gitclear.com/ai_assistant_code_quality_2025_research (Zugriff am 28. Februar 2025)
- [8] N. Friedman, "Introducing GitHub Copilot: your AI pair programmer", GitHub Inc., 2021. [Online]
Verfügbar: <https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/> (Zugriff am 28. Februar 2025)
- [9] Google Cloud, "DORA: 2024 Accelerate State of DevOps Report", Google Cloud, 2024. [Online]
Verfügbar: https://services.google.com/fh/files/misc/2024_final_dora_report.pdf (Zugriff am 01. März 2025)
- [10] IEEE, "1074-2006—IEEE Standard for Developing a Software Project Life Cycle Process", IEEE, New York City 2006.

- [11] ISO 9126-1, “Information technology — Software product quality — Part 1: Quality model”, ISO/IEC 9126-1:2001, International Organization for Standardization, Genf 2001.
- [12] R.S. Pressman, B. R. Maxim, “Software Engineering: A Practitioner's Approach”, McGraw Hill Higher Education, 9. Auflage, New York City 2019.
- [13] Digital Inc., “Case study: Finding defects earlier yields enormous savings”, Dulles 2007. [Online]
Verfügbar: <https://goo.gl/OQlyNi> (Zugriff am 03. Januar 2025)
- [14] B. Boehm, V. Basili, “Software Defect Reduction Top 10 List,” IEEE Computer, Vol. 34, No. 1, Pages 135–137, Washington 2001.
- [15] M. Anaya, “Clean Code in Python: Second Edition”, Packt Publishing Ltd, Birmingham 2020.
- [16] F Deißenböck, “Continuous Quality Control of Long-Lived Software Systems”, Technische Universität München, München 2010.
- [17] B. Kitchenham, S. L. Pfleeger, „Software quality: The elusive target”, IEEE Software, Vol. 13, No. 1, Pages 12-21, Washington 1996.
- [18] D. A. Garvin, “What does Product Quality really mean?”, MIT Sloan Management Review, Harvard 1984.
- [19] J. Voas, “Can clean pipes produce dirty water?”, IEEE Software, Vol. 14, No. 4, Pages 93-95, Washington 1997.
- [20] C. Jones, “Software assessments, benchmarks, and best practices.”, Addison-Wesley Longman Publishing Co, Inc., Boston 2000.
- [21] C. Lange, “Softwarequalitätsmodelle”, Technische Universität München, München 2010.
- [22] D. Hooker, “Internal and External Quality”, 2010. [Online]
Verfügbar: <https://wiki.c2.com/?InternalAndExternalQuality> (Zugriff am 01. Januar 2025)
- [23] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures”, University of California, Irvine 2000.
- [24] C. Mayer, “The Art of Clean Code: Best Practices to eliminate complexity and simplify your life”, No Starch Press, San Francisco 2022.

- [25] S. Freeman, N. Pryce, "Growing Object-Oriented Software, Guided by Tests", Addison-Wesley Professional 1st edition, London 2009.
- [26] D. Stavrinoudis, M. Xenos, "Comparing internal and external software quality measurements", University of Piraeus, Piraeus 2008.
- [27] B. W. Boehm, J. R. Brown, M. Lipow, "Quantitative Evaluation of Software Quality", Proceedings of the 2nd International Conference on Software Engineering, San Francisco 1976.
- [28] J. A. McCall, P. K. Richards, G. F. Walters, "Factors in Software Quality - Concept and Definitions of Software Quality", Rome Air Development Center, New York 1977.
- [29] R. B. Grady; D. L. Caswell, "Software Metrics: Establishing a Company-Wide Program", Prentice-Hall, Upper Saddle River 1992.
- [30] V. R. Basili, G. Caldiera, H. D. Rombach: "The goal question metric approach", Universität Kaiserslautern, Kaiserslautern 1994.
- [31] R. G. Dromey, K. Popper, "A Model for Software Product Quality", IEEE Trans. Software Eng. 21, Griffith 1995.
- [32] J. Bansya, C. G. Davis, "A hierarchical model for object-oriented design quality assessment", IEEE Trans. Software Eng. 28, Hayward 2002.
- [33] ISO 9126, "Software engineering — Product quality", ISO/IEC 9126:1991, International Organization for Standardization, Genf 1991.
- [34] ISO 9126, "Software engineering —Product quality", ISO/IEC 9126:2001, International Organization for Standardization, Genf 2001.
- [35] ISO 25010, "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models", ISO/IEC 25010:2011, International Organization for Standardization, Genf 2011.
- [36] ISO 25010, "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model", ISO/IEC 25010:2023, International Organization for Standardization, Genf 2023.
- [37] ISO 25059, "Software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality model for AI systems", ISO/IEC 25059:2023, International Organization for Standardization, Genf 2023.
- [38] G. Starke, „arc42 Quality Model – Q42“, arc42, Köln 2023. [Online]
Verfügbar: <https://quality.arc42.org/> (Zugriff am 14. Januar 2025)

- [39] K. Lochmann, S. Winter, A. Goeb, M. Kläs, and S. Nunnenmacher, "Software Quality Models in Practice – Survey Results", Technische Universität München, München 2012.
- [40] L. Lazic, N. Mastorakis, "Cost Effective Software Test Metrics", University of Novi Pazar, Novi Pazar 2008.
- [41] R. J. Rubey, R. D. Hartwick, "Quantitative measurement of program quality", Logicon Inc., San Pedro 1968.
- [42] D. R. McAndrews, "Establishing a software measurement process", Carnegie Mellon University - Software Engineering Institute, Pittsburgh 1993.
- [43] M. Broy, M. Kuhrmann, "Projektorganisation und Management im Software Engineering", Springer-Verlag Berlin, Heidelberg 2013.
- [44] M. Lacchia, "Radon: Introduction to Code Metrics", 2020. [Online]
Verfügbar: <https://radon.readthedocs.io/en/latest/intro.html> (Zugriff am 28. Januar 2025)
- [45] T. J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320, 1976.
- [46] ISO 5807, "Information processing — Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts", ISO/IEC 5807:1985, International Organization for Standardization, Genf 1985.
- [47] H. D. Mills, "Mathematical foundations for structured programming", IBM Federal System Division, FSC 72-6012., Gaithersburg 1972.
- [48] M. Lacchia, "Radon: Command-line Usage", 2020. [Online]
Verfügbar: <https://radon.readthedocs.io/en/latest/commandline.html> (Zugriff am 28. Januar 2025)
- [49] M. H. Halstead, "Elements of Software Science (Operating and programming systems series)", Elsevier Science Inc., New York 1977.
- [50] C. S. Thirumalai, H. Thirunavukkarasu, G. Vidhyagaran, K. Seenu, "Software Complexity Analysis Using Halstead Metrics", International Conference on Trends in Electronics and Informatics (ICEL), Tirunelveli 2017.
- [51] P. Omand, J. Hagemeister, "Metrics for assessing a software system's maintainability", Proceedings International Conference on Software Mainatenance (ICSM), pp. 337-344, Orlando 1992.

- [52] J. T. Foreman, J. Gross, R. Rosenstein, D. Fisher, K. Brune, "C4 Software Technology Reference Guide: A Prototype", Carnegie Mellon University - Software Engineering Institute, Pittsburgh 1997.
- [53] D. Coleman, "Using Metrics to Evaluate Software System Maintainability", Computer Vol. 27, No. 8, pp. 44-49, 1994.
- [54] D. Coleman, V. Lowther, P. Oman, "The Application of Software Maintainability Models in Industrial Software Systems", Journal of Systems Software Vol. 29, No. 1, pp. 3-16, 1995.
- [55] T. Pearse, P. Oman, "Maintainability Measurements on Industrial Source Code Maintenance Activities", Proceedings of the International Conference on Software Maintenance, Opio, Frankreich, 17.-20. Okt. 1995, pp. 295-303. Los Alamitos, CA: IEEE Computer Society Press, 1995.
- [56] K. D. Welker, P. Oman, "Software Maintainability Metrics Models in Practice", Journal of Defense Software Engineering Vol. 8, No. 11, pp. 19-23, 1995.
- [57] Microsoft Corporation, "Code metrics - Maintainability index range and meaning", Learn Code Analysis in Visual Studio 2022, Redmond 2007. [Online]
Verfügbar: <https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning> (Zugriff am 22. Februar 2025)
- [58] A. van Deursen, "Think Twice Before Using the Maintainability Index", Arie van Deursen Blog, 2014. [Online]
Verfügbar: <https://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/> (Zugriff am 22. Februar 2025)
- [59] D. I.K. Sjøberg, B. Anda, A. Mockus, "Questioning Software Maintenance Metrics: A Comparative Case Study", University of Oslo, 2012.
- [60] M. Riaz, M. Mendes, E.D. Tempero, "A Systematic Review of Software Maintainability Prediction and Metrics", 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM 2009), pp. 367-377, Orlando 2009.
- [61] D. Hooker, "Seven Principles of Software Development", 1996. [Online]
Verfügbar: <https://wiki.c2.com/?SevenPrinciplesOfSoftwareDevelopment> (Zugriff am 23. Februar 2025)
- [62] D. Hooker, "First Principle of Software Development: The Reason It All Exists", 1996. [Online]
Verfügbar: <https://wiki.c2.com/?TheReason> (Zugriff am 16. Januar 2025)

- [63] D. Hooker, “Definiton of Real Value”, 1996. [Online]
Verfügbar: <https://wiki.c2.com/?RealValue> (Zugriff am 16. Januar 2025)
- [64] D. Hooker, “Second Principle of Software Development: Keep It Short & Simple”, 1996. [Online]
Verfügbar: <https://wiki.c2.com/?KeepItSimple> (Zugriff am 16. Januar 2025)
- [65] D. Hooker, “Fourth Principle of Software Development: What You Produce, Others Will Consume”, 1996. [Online]
Verfügbar: <https://wiki.c2.com/?WhatYouProduceTheyConsume> (Zugriff am 16. Januar 2025)
- [66] D. Hooker, “Extrem Programming Practice: You Arent Gonna Need It”, 1996. [Online]
Verfügbar: <https://wiki.c2.com/?YouArentGonnaNeedIt> (Zugriff am 16. Januar 2025)
- [67] D. Hooker, “Fifth Principle of Software Development: Build for Today Design for Tomorrow”, 1996. [Online]
Verfügbar: <https://wiki.c2.com/?BuildForTodayDesignForTomorrow> (Zugriff am 16. Januar 2025)
- [68] S. Raschka, “Build a Large Language Model (From Scratch)”, Manning Publications Co, New York 2025.
- [69] V. Alto, “Building LLM Powered Applications: Create intelligent apps and agents with large language models”, Packt Publishing Ltd, Birmingham 2024.
- [70] Google Research, “The WordPiece Algorithm in Open Source BERT”, 2018. [Online]
Verfügbar: <https://github.com/google-research/bert/blob/master/tokenization.py#L300> (Zugriff am 28. Februar 2025)
- [71] A. Pandey, “Pytorch implementation of the GPT-1 Model”, 2022. [Online]
Verfügbar: <https://github.com/akshat0123/GPT-1> (Zugriff am 28. Februar 2025)
- [72] OpenAI, “The BBPE Algorithm in Open Source GPT-2”, 2019. [Online]
Verfügbar: <https://github.com/openai/gpt-2/blob/master/src/encoder.py#L42> (Zugriff am 28. Februar 2025)
- [73] OpenAI, “Tiktoken”, 2025. [Online]
Verfügbar: <https://github.com/openai/tiktoken> | https://github.com/openai/openai-cookbook/blob/main/examples/How_to_count_tokens_with_tiktoken.ipynb (Zugriff am 28. Februar 2025)
- [74] C. Wang, K. Cho, J. Gu, “Neural Machine Translation with Byte-Level Subwords”, Facebook AI Research, 2019.

- [75] Hugging Face, “Summary of the tokenizers”, 2025. [Online]
Verfügbar: https://huggingface.co/docs/transformers/en/tokenizer_summary (Zugriff am 28. Februar 2025)
- [76] B. Feng, “What is BBPE -- Tokenizer behind LLMs”, 2024. [Online]
Verfügbar: <https://www.kaggle.com/code/binfeng2021/what-is-bbpe-tokenizer-behind-lms> (Zugriff am 28. Februar 2025)
- [77] OpenAI, “Tokenizer: Learn about language model tokenization”, OpenAI Inc., 2025. [Online]
Verfügbar: <https://platform.openai.com/tokenizer> (Zugriff am 28. Februar 2025)
- [78] 3Blue1Brown, “Transformers (how LLMs work) explained visually | DL5“, 2024. [Online]
Verfügbar: https://youtu.be/wjZofJX0v4M?si=9_CPP9o8ImmlRM-&t=945 (Zugriff am 28. Februar 2025)
- [79] T. Mikolow, K. Chen, G. Corrado, J. Dean, “Efficient Estimation of Word Representations in Vector Space“, Mountain View 2013.
- [80] S. Matzka, “Künstliche Intelligenz in den Ingenieurwissenschaften: Maschinelles Lernen verstehen und bewerten”, Springer Fachmedien Wiesbaden GmbH, 2021.
- [81] F. van Veen, S. Leijnen, “The Neural Network Zoo”, 2019. [Online]
Verfügbar: <https://www.asimovinstitute.org/neural-network-zoo/> (Zugriff am 06. März 2025)
- [82] Google, “Machine Learning Glossary: Logits”, 2025. [Online]
Verfügbar: <https://developers.google.com/machine-learning/glossary/#logits> (Zugriff am 07. März 2025)
- [83] J. Berryman, A. Ziegler, “Prompt Engineering for LLMs: The Art and Science of Building Large Language Model–Based Applications”, O'Reilly Media Inc., Sebastopol 2024.
- [84] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, “Attention Is All You Need”, Advances in neural information processing systems, Vol. 30, 2017.
- [85] Aider, “Aider LLM Leaderboards”, 2025. [Online]
Verfügbar: <https://aider.chat/docs/leaderboards/> (Zugriff am 08. März 2025)
- [86] M. Ravkine, “CanAiCode Leaderboard”, 2025. [Online]
Verfügbar: <https://huggingface.co/spaces/mike-ravkine/can-ai-code-results> (Zugriff am 08. März 2025)

- [87] ProsusAI, "ProLLM Benchmarks", 2025. [Online]
 Verfügbar: <https://www.prollm.ai/leaderboard/stack-eval> (Zugriff am 08. März 2025)
- [88] Artificial Analysis Inc., "Independent analysis of AI models and API providers", 2025. [Online]
 Verfügbar: <https://artificialanalysis.ai/> (Zugriff am 08. März 2025)
- [89] T. Nguyen, E. Wong, "In-context Example Selection with Influences", University of Pennsylvania, Pennsylvania 2024.
- [90] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, P. Liang, "Lost in the Middle: How Language Models Use Long Contexts", Stanford University, Stanford 2023.
- [91] OpenAI, "Prompt Engineering: Enhance results with prompt engineering strategies.", OpenAI Inc., 2025. [Online]
 Verfügbar: <https://platform.openai.com/docs/guides/prompt-engineering> (Zugriff am 08. März 2025)
- [92] A. Sangani, "Prompt Engineering with Llama 2 & 3", Meta Platforms Inc., 2025. [Online]
 Verfügbar: <https://www.deeplearning.ai/short-courses/prompt-engineering-with-llama-2/> (Zugriff am 08. März 2025)
- [93] C. Wang, C. Almagor, "Build Long-Context AI Apps with Jamba", AI21 Labs, 2025. [Online]
 Verfügbar: <https://www.deeplearning.ai/short-courses/build-long-context-ai-apps-with-jamba/> (Zugriff am 08. März 2025)
- [94] J. Phoenix, M. Taylor, "Prompt Engineering for Generative AI: Future-Proof Inputs for Reliable AI Outputs", O'Reilly Media Inc., 2024.
- [95] T. B. Brown *et al.*, "Language Models are Few-Shot Learners", OpenAI Inc., 2020.
- [96] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, A. Chadha, "A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications", Indian Institute of Technology Patna, 2024.
- [97] J. He, M. Rungta, D. Koleczek, A. Sekhon, F. Wang, S. Hasan, "Does Prompt Formatting Have Any Impact on LLM Performance?", Massachusetts Institute of Technology, Cambridge 2024.
- [98] J. Gruber, "Markdown", 2004. [Online]
 Verfügbar: <https://daringfireball.net/projects/markdown/> (Zugriff am 06. Januar 2025)
- [99] M. Cone, "The Markdown Guide", 2020. ISBN: 979-8656504492

- [100] G. van Rossum, B. Warsaw, N. Coghlan, “PEP-8: Style Guide for Python Code”, 2001. [Online]
 Verfügbar: <https://www.python.org/dev/peps/pep-0008/> (Zugriff am 03. Dezember 2024)
- [101] T. Peters, “PEP-20: The Zen of Python”, 2004. [Online]
 Verfügbar: <https://www.python.org/dev/peps/pep-0020/> (Zugriff am 03. Dezember 2024)
- [102] K. Reitz, T. Schlusser, “The Hitchhiker’s Guide to Python: Best Practices for Development”, O'Reilly Media Inc., Sebastopol 2016.
- [103] C. Mayer, “The Art of Clean Code: Best Practices to eliminate complexity and simplify your life”, No Starch Press, San Francisco 2022.
- [104] P. Silen, “Clean Code Principles and Patterns: Python Edition”, Lean Publishing, Victoria 2024.
- [105] A. Sweigart, “Beyond the Basic Stuff with Python: Best Practices for writing Clean Code”, No Starch Press Inc, San Francisco 2021.
- [106] G. A. Miller, „The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information”, Harvard University - Psychological Review, Vol. 101, No. 2, pp. 343-352, Harvard 1956.
- [107] N. Cowan, „The Magical Number 4 in Short-Term Memory: A Reconsideration of Mental Storage Capacity”, University of Missouri - Behavioral and Brain Sciences, Vol. 24, No. 1, pp. 87–114, Missouri 2001.
- [108] T. Peters, “PEP-282: A Logging System”, 2002. [Online]
 Verfügbar: <https://www.python.org/dev/peps/pep-0282/> (Zugriff am 06. Dezember 2024)
- [109] OWASP, “Application Logging Vocabulary Cheat Sheet”, 2025. [Online]
 Verfügbar:
https://cheatsheetseries.owasp.org/cheatsheets/Logging_Vocabulary_Cheat_Sheet.html (Zugriff am 06. Dezember 2024)
- [110] Python Software Foundation, “String: Format Specification Mini-Language”, 2025. [Online]
 Verfügbar: <https://docs.python.org/3/library/string.html#formatspec> (Zugriff am 06. Dezember 2024)

- [111] W. Xie, X. Peng, M. Liu, C. Treude, Z. Xing, X. Zhang, W. Zhao, "API method recommendation via explicit matching of functionality verb phrases", 2020.
- [112] IEEE, "754-2019-IEEE Standard for Floating-Point Arithmetic", IEEE, New York City 2019.
- [113] Python Software Foundation, "Floating-Point Arithmetic: Issues and Limitations", 2024. [Online]
Verfügbar: <https://docs.python.org/3/tutorial/floatingpoint.html> (Zugriff am 03. Dezember 2024)
- [114] G. van Rossum, J. Lehtosalo, L. Langa, "PEP 484: Type Hints", 2014. [Online]
Verfügbar: <https://www.python.org/dev/peps/pep-0484/> (Zugriff am 07. Dezember 2024)
- [115] J. Lehtosalo, "Mypy 1.13.0 Documentation: Type hints cheat sheet", 2022. [Online]
Verfügbar: https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html (Zugriff am 07. Dezember 2024)
- [116] D. Goodger, G. van Rossum, "PEP-257: Docstring Conventions", 2001. [Online]
Verfügbar: <https://www.python.org/dev/peps/pep-0257/> (Zugriff am 07. Dezember 2024)
- [117] Pandas via NumFOCUS Inc., "pandas docstring guide", 2024. [Online]
Verfügbar: https://pandas.pydata.org/docs/development/contributing_docstring.html (Zugriff am 10. Dezember 2024)
- [118] D. Goodger, "PEP-287: reStructuredText Docstring Format", 2023. [Online]
Verfügbar: <https://peps.python.org/pep-0287/> (Zugriff am 10. Dezember 2024)
- [119] Numpy via NumFOCUS Inc., "Documenting classes", 2024. [Online]
Verfügbar: <https://numpydoc.readthedocs.io/en/latest/format.html#documenting-classes> (Zugriff am 10. Dezember 2024)
- [120] Numpy via NumFOCUS Inc., "Documenting modules", 2024. [Online]
Verfügbar: <https://numpydoc.readthedocs.io/en/latest/format.html#documenting-modules> (Zugriff am 11. Dezember 2024)
- [121] Ł. Langa, "Black: The Uncompromising Code Formatter", 2025. [Online]
Verfügbar: <https://black.readthedocs.io/en/stable/> (Zugriff am 08. Dezember 2024)
- [122] H. Hattori, "autopep8: A tool that automatically formats Python code to conform to the PEP 8 style guide", 2025. [Online]
Verfügbar: <https://pypi.org/project/autopep8/> (Zugriff am 09. Dezember 2024)

- [123] Anthropic, “API Pricing & Rate Limits”, 2025. [Online]
Verfügbar: <https://www.anthropic.com/pricing#anthropic-api> |
<https://docs.anthropic.com/en/api/rate-limits#tier-1> (Zugriff am 08. März 2025)
- [124] Google Gemini, “API Pricing & Rate Limits”, 2025. [Online]
Verfügbar: <https://ai.google.dev/gemini-api/docs/pricing> |
<https://ai.google.dev/gemini-api/docs/rate-limits#tier-1> (Zugriff am 08. März 2025)
- [125] OpenAI, “API Pricing & Rate Limits”, 2025. [Online]
Verfügbar: <https://platform.openai.com/docs/models/gpt-4o#tier-1> (Zugriff am 08. März 2025)
- [126] Xai, “API Pricing & Rate Limits”, 2025. [Online]
Verfügbar: <https://docs.x.ai/docs/models?cluster=us-east-1#models-and-pricing> |
<https://docs.x.ai/docs/consumption-and-rate-limits> (Zugriff am 08. März 2025)
- [127] C. Sadowski, E. Söderberg, L. Church, M. Sipko, A. Bacchelli, “Modern Code Review: A Case Study at Google“, IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), Gothenburg 2018.
- [128] GitHub Inc, “Pricing – Plans for every developer”, 2025. [Online]
Verfügbar: <https://github.com/pricing> (Zugriff am 25. März 2025)
- [129] Wirtschaftskammer Österreich, “Klein- und Mittelbetriebe in Österreich: Was versteht man unter KMU in der Statistik?”, 2024. [Online]
Verfügbar: <https://www.wko.at/zahlen-daten-fakten/kmu-definition> (Zugriff am 26. März 2025)
- [130] S. Kumar, B. Bonnefoy-Claudet “python-dotenv 1.0.1 Documentation”, 2024. [Online]
Verfügbar: <https://pypi.org/project/python-dotenv/> (Zugriff am 08. März 2025)
- [131] A. Wiggins, “The Twelve-Factor App”, Heroku from Salesforce, San Francisco 2017. [Online]
Verfügbar: <https://12factor.net/> (Zugriff am 08. März 2025)

Abbildungsverzeichnis

Abbildung 1: Aufbau dieser Arbeit	11
Abbildung 2: Beliebtheit von Programmiersprachen basierend auf dem PYPL-Index (Februar 2025) [1] (eigene Abbildung)	12
Abbildung 3: Beliebtheit von Programmiersprachen basierend auf dem TIOBE-Index (Februar 2025) [2] (eigene Abbildung)	13
Abbildung 4: Google-Suchinteresse an Python und MATLAB weltweit (2004-2025) [4] (eigene Abbildung)	14
Abbildung 5: Anteil der Stack Overflow-Fragen zu Python und MATLAB pro Monat (2008–2025) [5] (eigene Abbildung)[1]	14
Abbildung 6: Relative Verteilung der Code-Änderungen pro Jahr, einschließlich einer Prognose für 2025 von GitClear [7] (eigene Abbildung).....	16
Abbildung 7: (Nicht inflationsbereinigte) Relative Kosten für die Korrektur eines Fehlers [13][14] (eigene Darstellung)	19
Abbildung 8: Aufbau eines Reifegradmodells (Capability Maturity Model)	21
Abbildung 9: Einfluss der Codequalität (innere Qualität) auf den Entwicklungsprozess [24] .	22
Abbildung 10: Relevanz der inneren und äußeren Qualität während den Testphasen [25] (eigene Darstellung)	23
Abbildung 11: Zusammenhang zwischen innerer und äußerer Qualität nach [26].....	24
Abbildung 12: Zeitstrahl der Historie der relevantesten Qualitätsmodelle	25
Abbildung 13: Qualitätseigenschaften nach Boehm [27] (eigene Abbildung)	26
Abbildung 14: Qualitätseigenschaften nach McCall [28] (eigene Abbildung).....	29
Abbildung 15: Qualitätseigenschaften nach FURPS [29] (eigene Abbildung)	30
Abbildung 16: Systematische Darstellung des Goal Question Metric-Modells [30] (eigene Abbildung)	32
Abbildung 17: Grundgedanke des Dromey Qualitätsmodells [31] (eigene Abbildung)	33
Abbildung 18: Beispielhafte Anwendung des Dromey Qualitätsmodells [31] (eigene Abbildung)	35
Abbildung 19: Qualitätseigenschaften von Softwaresystemen nach ISO/IEC 9126-1:2001 [33] (eigene Abbildung)	36
Abbildung 20: Qualitätseigenschaften von Softwaresystemen nach ISO/IEC 25010:2011 [35] (eigene Abbildung)	37
Abbildung 21: Qualitätseigenschaften von Softwaresystemen nach ISO/IEC 25010:2023 [36] (eigene Abbildung)	39
Abbildung 22: Qualitätseigenschaften von AI-Softwaresystemen nach ISO/IEC 25059:2023 [37] (eigene Abbildung)	40
Abbildung 23: Herkunft und Unternehmensgröße der Befragten [39].....	42
Abbildung 24: Praxiseinsatz von Qualitätsmodellen [39].....	42
Abbildung 25: Anpassung der Qualitätsmodelle an individuelle Anforderungen [39].....	43

Abbildung 26: Nutzungs frequenz der verwendeten Qualitätssicherungsmethoden [39]	43
Abbildung 27: Die relevantesten Qualitätsmerkmale mit Verbesserungspotenzial [39]	44
Abbildung 28: Kontrollflussdiagramm G des gegebenen Python-Skriptes	49
Abbildung 29: Hierarchische Struktur der Künstlichen Intelligenz [68]	60
Abbildung 30: Vom Pretraining bis zum Fine-Tuning eines LLMs [68]	60
Abbildung 31: Vorhersage des wahrscheinlichsten nächsten Wortes in einem LLM [69] (eigene Abbildung)	61
Abbildung 32: Darstellung verschiedener Token-Embeddings als Vektoren [78]	63
Abbildung 33: Vergleich zwischen Feed Forward Neural Networks und Recurrent Neural Networks [80] (eigene Abbildung)	64
Abbildung 34: Einfluss der Temperatur auf die Wahrscheinlichkeitsverteilung des nächsten Tokens [68] (eigene Darstellung)	65
Abbildung 35: Einfluss der Temperatur auf die Ausgabe eines LLMs (OpenAI's text-davinci-003) [83]	66
Abbildung 36: Entscheidungsdiagramm zur Notwendigkeit von Fine-Tuning [83]	67
Abbildung 37: Anwendungsfelder von LLM-basierten Applikationen [83]	68
Abbildung 38: Ablauf der Entwicklung eines LLM-Workflows [83]	69
Abbildung 39: Vergleich der Struktur eines Zero-Shot-Prompts mit der eines Few-Shot-Prompts [83]	72
Abbildung 40: Einfluss von Modellgröße und Prompting auf die Genauigkeit bei der Entfernung von zufälligen Symbolen aus Wörtern [95]	72
Abbildung 41: Eigenschaften von namhaften Python-Projekten [68]	76
Abbildung 42: Ablaufdiagramm des "All-at-Once"- und des "One-by-One"-Ansatzes	126
Abbildung 43: Ablaufdiagramm des "All-at-Once"- und des "One-by-One"-Ansatzes	138
Abbildung 44: Vergleich der Zuverlässigkeit verschiedener LLMs und Strategien über drei Fallstudien hinweg	140
Abbildung 45: Vergleich der funktionalen Korrektheit verschiedener LLMs und Strategien über drei Fallstudien hinweg	140
Abbildung 46: Vergleich der Pylint-Scores verschiedener LLMs und Strategien über drei Fallstudien hinweg	141
Abbildung 47: Vergleich der Zeilenanzahl verschiedener LLMs und Strategien über drei Fallstudien hinweg	141
Abbildung 48: Vergleich der durchschnittlichen Funktionslänge (mit und ohne Docstrings) verschiedener LLMs und Strategien über drei Fallstudien hinweg	142
Abbildung 49: Vergleich der durchschnittlichen zyklomatischen Komplexität verschiedener LLMs und Strategien über drei Fallstudien hinweg	142
Abbildung 50: Vergleich des Maintainability-Indexes verschiedener LLMs und Strategien über drei Fallstudien hinweg	143
Abbildung 51: Vergleich des Tokenverbrauchs verschiedener LLMs und Strategien über drei Fallstudien hinweg	143

Abbildung 52: Vergleich der Optimierungskosten verschiedener LLMs und Strategien über alle drei Fallstudien hinweg.....	144
Abbildung 53: Vergleich der Optimierungsdauer verschiedener LLMs und Strategien über alle drei Fallstudien hinweg.....	145
Abbildung 54: Vergleich der Regelverstöße verschiedener LLMs und Strategien über alle drei Fallstudien hinweg.....	146
Abbildung 55: Durchschnittliche Anzahl an Regelverstößen über alle Ergebnisse der Modelle claude-3.7-sonnet-20250219, gemini-2.0-pro-exp-02-05 und gpt-4o-2024-08 hinweg.....	147
Abbildung 56: Durchschnittliche Optimierungsdauer über alle Ergebnisse der Modelle claude-3.7-sonnet-20250219, gemini-2.0-pro-exp-02-05 und gpt-4o-2024-08 hinweg	147
Abbildung 57: Durchschnittliche Optimierungskosten über alle Ergebnisse der Modelle Claude-3.7-Sonnet-20250219, Gemini-2.0-Pro-Exp-02-05 und GPT-4o-2024-08 hinweg...148	
Abbildung 58: Durchschnittliche Anzahl an Regelverstößen über alle Fallstudien hinweg ..150	
Abbildung 59: Arbeitgeberbruttoersparnis pro Entwickler:in pro Jahr in Abhängigkeit der prozentualen Zeitersparnis durch OptiPy auf Basis eines monatlichen Arbeitnehmerbruttos von 4500€ ..156	
Abbildung 60: Arbeitgeberbruttoersparnis pro Jahr in Abhängigkeit der prozentualen Zeitersparnis durch OptiPy auf Basis eines monatlichen Arbeitnehmerbruttos von 4500€..156	

Tabellenverzeichnis

Tabelle 1: Einsatzgebiete von Python im Maschinenbau	15
Tabelle 2: Gelieferte Fehler pro Funktionsblock in Bezug zu CMM-Stufe [20]	21
Tabelle 3: Beispielhafte Anwendung des GQM-Ansatzes [30]	32
Tabelle 4: Beziehung zwischen Hauptqualitätsmerkmalen und qualitätstragenden Eigenschaften [31].....	34
Tabelle 5: Beziehung zwischen Hauptqualitätsmerkmalen und qualitätstragenden Eigenschaften [31].....	34
Tabelle 6: Historie der relevantesten Software-Qualitätsmodelle	45
Tabelle 7: Einfluss verschiedener Programmkonstrukten auf die zyklomatische Komplexität [44].....	51
Tabelle 8: Interpretation der zyklomatischen Komplexität [48]	52
Tabelle 9: Interpretation des Maintainability Index [48]	55
Tabelle 10: Übersicht über Tokenizer-Typen, ihre Anwendungen und Beispiele	62
Tabelle 11: Einfluss der Temperatur auf die Generierung von Texten [83]	65
Tabelle 12: Vergleich von verschiedenen Fine-Tuning-Methoden [83].....	67
Tabelle 13: Vergleich zwischen LLM-Agenten und LLM-Workflows	68
Tabelle 14: Vergleich der verwendeten LLMs.....	125
Tabelle 15: Auswertung der Fallstudie 1 vor der Optimierung.....	128
Tabelle 16: Auswertung der Fallstudie 2 vor der Optimierung.....	131
Tabelle 17: Auswertung der Fallstudie 3 vor der Optimierung.....	134
Tabelle 18: Durchschnittliche Anzahl an Regelverstößen, Optimierungsdauer, Optimierungskosten über alle Ergebnisse der Modelle Claude-3.7-Sonnet-20250219, Gemini-2.0-Pro-Exp-02-05 und GPT-4o-2024-08 hinweg	148
Tabelle 19: Nutzwertanalyse der Optimierungsstrategien der ersten Fallstudie über alle Modelle hinweg	149
Tabelle 20: Nutzwertanalyse der Optimierungsstrategien der zweiten Fallstudie über alle Modelle hinweg	149
Tabelle 21: Nutzwertanalyse der Optimierungsstrategien der dritten Fallstudie über alle Modelle hinweg	149
Tabelle 22: Durchschnittliche Anzahl an Regelverstößen, Optimierungsdauer, Optimierungskosten des gpt-4o-2024-08-06 Modells	150
Tabelle 23: Nutzwertanalyse der Optimierungsstrategien der ersten Fallstudie für gpt-4o-2024-08-06	151
Tabelle 24: Nutzwertanalyse der Optimierungsstrategien der zweiten Fallstudie für gpt-4o-2024-08-06.....	151
Tabelle 25: Nutzwertanalyse der Optimierungsstrategien der dritten Fallstudie für gpt-4o-2024-08-06.....	151

Tabelle 26: Berechnung der durchschnittlichen Optimierungskosten pro Zeile für das gpt-4o-2024-08-06 Modells in Kombination mit dem „One-by-One“-Ansatz	153
Tabelle 27: Wirtschaftlichkeitsvergleich der eingesparten Arbeitgeberbruttokosten pro Entwickler:in pro Jahr durch OptiPy in einem Kleinunternehmen mit 10 Entwickler:innen...	155
Tabelle 28: Wirtschaftlichkeitsvergleich der eingesparten Arbeitgeberbruttokosten pro Entwickler:in pro Jahr durch OptiPy in einem mittleren Unternehmen mit 100 Entwickler:innen	155
Tabelle 29: Wirtschaftlichkeitsvergleich der eingesparten Arbeitgeberbruttokosten pro Entwickler:in pro Jahr durch OptiPy in einem Großunternehmen mit 1000 Entwickler:innen	155

Dokumentationstabelle KI-basierte Hilfsmittel

KI-basierte Hilfsmittel	Verwendungszweck	Prompt, Quelle, Seite, Absatz ...
DeepL Translate	Teilweise Übersetzung der Kurzfassung in die englische Sprache	Abstract (Seite 4)
ChatGPT-4o	Überprüfung der Grammatik und Rechtschreibung	"Please list issues with spelling and grammar in the following text: ..." Attachment: Thesis as .pdf

Abkürzungsverzeichnis

API	Application Programming Interface
CC	Cyclomatic Complexity
DNN	Deep Neural Network
FFNN	Feed Forward Neural Network
FURPS	Functionality / Usability / Reliability / Performance / Supportability
GenAI	Generative AI
GQM	Goal Question Metric
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
KI	Künstliche Intelligenz
KISS	Keep It Short & Simple
LLM	Large Language Model
LOC	Lines Of Code
LSTM	Long Short-Term Memory Network
MI	Maintainability Index
NLP	Natural Language Processing
PeFT	Parameter-Efficient Fine-Tuning
PEP	Python Enhancement Proposal
REST	Representational State Transfer
RNN	Recurrent Neural Network
SLOC	Source Lines Of Code
YAGNI	You Aren't Gonna Need It
YAML	Yet Another Markup Language

Anhang A: guideline.md

```
# Clean Code Guideline

## 1. General Pythonic Practices

#### f-Strings
- Use f-Strings instead of String Concatenation/filter()
```python
WRONG
print(name + " is " + str(age) + " years old.")
```
```python
BETTER
print("{} is {} years old.".format(name, age))
```
```python
BEST
print(f"{name} is {age} years old.")
```

#### Use Logging instead of print()-Statements
- Define the logging.basicConfig in the module level (at the top of the file, after imports) and not within `if __name__ == "__main__":` block.
```python
WRONG
def divide(dividend, divisor):
 print(f"Received input: dividend={dividend}, divisor={divisor}")

 if divisor == 0:
 print("Error: Division by zero attempted!")
 raise ValueError("Division by zero is not allowed.")

 result = dividend / divisor
 print(f"Calculation successful: {dividend} / {divisor} = {result}")
 return result
```
```python
CORRECT
import logging

logging.basicConfig(
 level=logging.INFO,
 format=[
 "[%(asctime)s.%03d][%(levelname)s]",
 "[%(filename)s:%(lineno)d - %(funcName)s(): %(message)s"
],
 datefmt="%d-%m-%Y %H:%M:%S",
)
```

def divide(dividend, divisor):
    logging.info(f"Received input: {dividend}, {divisor}")

    if divisor == 0:
        logging.error("Error: Division by zero attempted!")
        raise ValueError("Division by zero is not allowed.")

    result = dividend / divisor
    logging.info(f"Calculation successful: {dividend} / {divisor} = {result}")
    return result
```

```

```
- Use f-strings within the logging command, not %s.

List Comprehensions
- Simplify for-Loops with List Comprehensions
```python
# WRONG
even_numbers = []
for number in numbers:
    if number % 2 == 0:
        even_numbers.append(number)
```
```python
# CORRECT
even_numbers = [number for number in numbers if number % 2 == 0]
```

Context Managers
- Using Context Managers when accessing files
```python
# WRONG
input_file = open("input.txt", "r")
content = input_file.read()
input_file.close()
```
```python
# BETTER
with open("input.txt", "r") as input_file:
    content = input_file.read()
```

- Always specify encoding to avoid decoding issues
```python
# BEST
with open("input.txt", "r", encoding="utf-8") as input_file:
    content = input_file.read()
```

When accessing an index use `enumerate(elements)` instead of `range(len(elements))`
```python
# WRONG
for i in range(len(elements)):
    print(f"Index: {i}, Element: {elements[i]}")
```
```python
# CORRECT
for i, element in enumerate(elements):
    print(f"Index: {i}, Element: {element}")
```

2. Refactoring

Avoid deep nesting by refactoring code into functions/methods with a maximum of 3 indentation levels
```python
# WRONG
def check_permissions(user):
    if user:
        if user.is_valid():
            if user.has_role("admin"):
                if user.has_permission("edit"):
                    print("Access granted!")
```

```

```

 else:
 print("Permission denied!")
 else:
 print("User has no admin role!")
 else:
 print("User is invalid!")
 else:
 print("No user provided!")
```
```python
CORRECT
def check_permissions(user):
 if not user:
 print("No user provided!")
 return

 if not user.is_valid():
 print("User is invalid!")
 return

 if not user.has_role("admin"):
 print("User has no admin role!")
 return

 if not user.has_permission("edit"):
 print("Permission denied!")
 return

 print("Access granted!")
```

### Refactor long functions into smaller ones with a maximum line count of 10
- Reduce complexity & Improve maintainability
```
```python
# WRONG
def process_order(order):
    total = sum(item['price'] * item['quantity'] for item in order['items'])

    if total > 100:
        discount = total * 0.1
    else:
        discount = 0

    final_price = total - discount

    print(f"Final Price: {final_price}")
```
```
```python
CORRECT
def calculate_total(order):
 return sum(item['price'] * item['quantity'] for item in order['items'])

def apply_discount(total):
 return total * 0.1 if total > 100 else 0

def process_order(order):
 total = calculate_total(order)
 discount = apply_discount(total)
 final_price = total - discount
 print(f"Final Price: {final_price}")
```

### One Statement per line

```

```

- This accounts for assignments too:
```python
WRONG
customer_name = "Albert Einstein", customer_id = 1234
```
```python
CORRECT
customer_name = "Albert Einstein"
customer_id = 1234
```

- This accounts for bool-assignments and if-statements:
```python
WRONG
if number > 0 and number % 2 == 0 and number % 99 == 0: print(f"{number} is valid!")
```
```python
CORRECT
is_positive = number > 0
is_even = number % 2 == 0
is_divisible_by_ninety-nine = number % 99 == 0

if is_positive and is_even and is_divisible_by_ninety-nine:
 print(f"{number} is valid!")
```

### Remove duplicate code/functions/methods/classes
```python
WRONG
print("Good morning!")
print("How's your mood right now?")
mood = input()
print(f"Glad to hear that you feel {mood}.")

print("Good afternoon!")
print("How's your mood right now?")
mood = input()
print(f"Glad to hear that you feel {mood}.")

print("Good evening!")
print("How's your mood right now?")
mood = input()
print(f"Glad to hear that you feel {mood}.")
```
```python
BETTER
def check_mood():
 print("How's your mood right now?")
 mood = input()
 print(f"Glad to hear that you feel {mood}.")

print("Good morning!")
check_mood()

print("Good afternoon!")
check_mood()

print("Good evening!")
check_mood()
```
```python
BEST
def check_mood(time_of_day):

```

```

print(f"Good {time_of_day}!")
print("How's your mood right now?")
mood = input()
print(f"Glad to hear that you feel {mood}.")
```

**times\_of\_day = ["morning", "afternoon", "evening"]**

```

for time_of_day in times_of_day:
 check_mood(time_of_day)
```

```

Remove dead/commented out code

- Remove dead code commented out with Comments (#)

```

```python
WRONG
def calculate_total(price, discount):
 # total = price + (price * discount)
 return price * (1 + discount)
```

```

```

```python
CORRECT
def calculate_total(price, discount):
 return price * (1 - discount)
```

```

Refactor the code to ensure no global variables are used

```

```python
WRONG
def read_api_key() -> None:
 load_dotenv(override=True)
 global API_KEY
 API_KEY = os.getenv("API_KEY")
```

```

```

```python
CORRECT
def read_api_key() -> str:
 load_dotenv(override=True)
 API_KEY = os.getenv("API_KEY")
 return API_KEY
```

```

Remove unnecessary comments (except module categories `# Standard library imports`, `# Third party imports`, `# Local imports`)

- ```python
- # WRONG

```

original_price = 1000 # Original price of the product
discount_rate = 0.1 # Discount rate (0.1 = 10%)

# Calculate the discounted price for the current year
discounted_price = original_price * (1 - discount_rate)

# Print the discounted price
print(f"{discounted_price}$")
```

```

```

```python
# CORRECT
original_price = 1000
discount_rate = 0.1

discounted_price = original_price * (1 - discount_rate)
print(f"{discounted_price}$")
```

```

```

3. Naming Conventions
- APPLY all following naming conventions
- Make sure no variable is defined twice to avoid type hinting issues

General Naming Rules (recommendation by PEP-8)
- All words should be english and consist of ASCII letters (no accent marks).
  ```python
  # WRONG
  öffnungszeiten = {...}
  straße = "..."
  ```

  ```python
  # CORRECT
  opening_hours = {...}
  street = "..."
  ```

- Use short, common names for clarity.
  ```python
  # WRONG
  def retrieve_data():
    ...
  ```

  ```python
  # CORRECT
  def get_data():
    ...
  ```

- Pick one term and use it consistently (e.g. `data`).
  ```python
  # WRONG
  def get_data():
    # ...

  def process_info():
    # ...

  def save_content():
    # ...
  ```

  ```python
  # CORRECT
  def get_data():
    # ...

  def process_data():
    # ...

  def save_data():
    # ...
  ```

- Avoid meaningless names & uncommon abbreviations (If necessary, use abbreviations, when longer than 20 characters, e.g. `very_very_long_string` → `very_very_long_str`)
  ```python
  # WRONG
  cstmrss = [...]
  nbrs = [...]
  temp_var = {...}
  ```

  ```python
  # CORRECT
  ```


```

```

customers = [...]
numbers = [...]
temperature_variance = {...}
```

- Use descriptive names to clarify functionality.
```python
WRONG
def f(...):
 # ...
```
```python
CORRECT
def eur_to_usd(...):
 # ...
```

- Avoid ambiguous names by specifying context.
```python
WRONG
def euro_to_dollar(...):
 # ...
```
```python
CORRECT
def eur_to_usd(...):
 # ...
```

- Generally avoid appending types to names unless necessary for clarity.
```python
WRONG
customer_list = [...]
country_to_currency_dict = {...}
PI_value = 3.14
```
```python
CORRECT
customers = [...]
country_to_currency = {...}
PI = 3.14
```

    Except: In some cases it makes sense to append variable type, but only when differentiating between multiple representations of the same data.
    _str, _int, _float, _list, _dict, _set
```python
CORRECT
age_int = 33
age_str = str(age_int)
```

- Don't Overwrite Built-in Names: ArithmeticError, AssertionError, AttributeError, BaseException, BaseExceptionGroup, BlockingIOError, BrokenPipeError, BufferError, BytesWarning, ChildProcessError, ConnectionAbortedError, ConnectionError, ConnectionRefusedError, ConnectionResetError, DeprecationWarning, EOFError, Ellipsis, EncodingWarning, EnvironmentError, Exception, ExceptionGroup, False, FileExistsError, FileNotFoundError, FloatingPointError, FutureWarning, GeneratorExit, IOError, ImportError, ImportWarning, IndentationError, IndexError, InterruptedError, IsADirectoryError, KeyError, KeyboardInterrupt, LookupError, MemoryError, ModuleNotFoundError, NameError, None, NotADirectoryError, NotImplemented, NotImplemented, OSError, OverflowError, PendingDeprecationWarning, PermissionError, ProcessLookupError, RecursionError, ReferenceError, ResourceWarning, RuntimeError, RuntimeWarning, StopAsyncIteration, StopIteration, SyntaxError, SyntaxWarning, SystemError, SystemExit, TabError, TimeoutError, True, TypeError, UnboundLocalError, UnicodeDecodeError, UnicodeEncodeError, UnicodeError,

```

```

UnicodeTranslateError, UnicodeWarning, UserWarning, ValueError, Warning, WindowsError,
ZeroDivisionError, __build_class__, __debug__, __doc__, __import__, __loader__, __name__,
__package__, __spec__, abs, aiter, all, anext, any, ascii, bin, bool, breakpoint,
bytearray, bytes, callable, chr, classmethod, compile, complex, copyright, credits,
delattr, dict, dir, divmod, enumerate, eval, exec, exit, filter, float, format, frozenset,
getattr, globals, hasattr, hash, help, hex, id, input, int, isinstance, issubclass, iter,
len, license, list, locals, map, max, memoryview, min, next, object, oct, open, ord, pow,
print, property, quit, range, repr, reversed, round, set, setattr, slice, sorted,
staticmethod, str, sum, super, tuple, type, vars, zip
    ```python
 # WRONG
 class Order:
 def __init__(self, id):
 self.id = id
    ```
    ```python
 # CORRECT
 class Order:
 def __init__(self, id_):
 self.id_ = id_
    ```

### Naming Modules
- Modules should be written in lowercase snake_case
- If necessary, use underscores to increase readability.
    ```python
 # WRONG
 import SomeModule
 import someModule
 import Somemodule
 import somemodeule
 import Some_Module
 import some_Module
 import Some_module
    ```
    ```python
 # CORRECT
 import module
 import some_other_module
    ```

```

```

### Naming Constants
- Constant variables should be written in uppercase SNAKE_CASE.
- Use precisely named constants to avoid magic numbers.
    ```python
 # WRONG
 profit_in_usd = 0.9 * profit_in_eur
    ```
    ```python
 # CORRECT
 EUR_TO_USD_RATE = 0.9
 profit_in_usd = EUR_TO_USD_RATE * profit_in_eur
    ```

```

```

### Naming Integer variables
- Integer variables should be written in lowercase snake_case.
- Most common: `number_of_<something>` or `<something>_count` (e.g. `number_of_failures` or `failure_count`)
- If unit not self-evident, add it at the end (e.g. `retry_delay` → `retry_delay_in_ms`)

    ```python
 # WRONG
 failures = 10
    ```

```

```

retry_delay = 500
distance = 33
price = 33
```
```python
# CORRECT
failure_count = 10
retry_delay_in_ms = 500
distance_in_km = 33
price_in_eur = 33
```

Naming Float variables
- Float variables should be written in lowercase snake_case.
- Most common: `<something>_amount` (e.g. `rainfall_amount`), use Int for money to avoid rounding errors.
- If unit not self-evident, add it at the end
 - `rainfall_amount_in_mm`
 - `angle_in_degrees` (values 0-360)
 - `failure_percent` (values 0-100)
 - `failure_ratio` (values 0-1)

Naming Boolean variables
- Boolean variables should be written in lowercase snake_case.
- The verb should be in the middle of boolean variable.
- Extract Constant for Boolean Expression
- Most common:
 - `is_<something>` → e.g. `is_disabled`
 - `has_<something>` → e.g. `has_errors`
 - `did_<something>` → e.g. `did_update`
 - `should_<something>` → e.g. `should_update`
 - `will_<something>` → e.g. `will_update`
- **No-Go: `<passive-verb>_something` (f.e.: `inserted_field` → `field_was_instered` verb should be in the middle of boolean)**

```
```python
WRONG
if is_pool_full:
 # ...
```
```python
CORRECT
if pool_is_full:
 # ...
```
```
```
```
```python
# WRONG
if inserted_table:
    # ...
```
```
```
```
```python
CORRECT
if table_was_instered:
 # ...
```
```
```
```
```python
# WRONG
machine_was_started = machine.start()
if not machine_was_started:
    # ...
```

```

```

```python
# CORRECT
machine_was_not_started = not machine.start()
if machine_was_not_started:
    # ...
```

- Extract boolean variables from constants, so if-statement only consist of explicit boolean variables
```python
# WRONG
if person.age > 18 and person.has_license:
    is_allowed_to_drive = True
```
```python
# CORRECT
is_adult = person.age > 18
has_license = person.has_license
is_allowed_to_drive = is_adult and has_license
```

Naming String variables
- String variables should be written in lowercase snake_case.
```python
# WRONG
UserName = "Amadeus"
```
```python
# CORRECT
user_name = "Amadeus"
```
- Consistent usage of quotation marks
```python
# WRONG
user_names = ["Amadeus", 'Ludwig']
```
```python
# CORRECT
user_names = ["Amadeus", "Ludwig"]
```

Naming Collection (List and Set) Variables
- Collection variables should be written in lowercase snake_case.
- Use plural form of noun for lists and sets (e.g. customers, tasks)
- Avoid embedding type (e.g., customer_list, task_set) :
```python
# WRONG
customer_list = [...]
student_id_set = {"001", "002", "003"}
```
```python
# CORRECT
customers = [...]
student_ids = {"001", "002", "003"}
```
unless necessary in some edge cases (e.g. queue_of_tasks) or in the following example:
```python
fruits_string = "banana apple cherry banana"
fruits_list = fruits_string.split(" ")
fruits_set = set(fruits_list)
```

```

```

Naming Dictionary Variables
- Dictionary variables should be written in lowercase snake_case
- Dictionary variables should follow the pattern `key_to_value` or `key_to_values`
```python
country_currencies = {
    "AT": "EUR",
    ...
}
```
```python
# CORRECT
country_to_currency = {
    "AT": "EUR",
    ...
}
```

```python
# WRONG
country_currencies = [
    "AT": ["EUR", "USD"],
    ...
]
```
```python
# CORRECT
country_to_currencies = {
    "AT": ["EUR", "USD"],
    ...
}
```

Naming Pair and Tuple Variables
- Tuples variables should be written in lowercase snake_case.
- If variable not needed use throwaway variable `_`.
```python
# CORRECT
height_and_width_in_mm = (100, 200)
coordinates = (48.2394, 16.3778)
```
- If variable not needed use throwaway variable `_`.
```python
# CORRECT
locations = [
    ("Vienna", 48.2394, 16.3778),
]

for city, _, _ in locations:
    print(city)
```

Naming Classes
- Class names should be written in PascalCase.
- Private attributes & methods should always begin with an underscore _.
- Public attributes & methods should never begin with an underscore _.
```python
# CORRECT
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self._balance = balance

    def get_balance(self):
```

```

```

 return self._balance

 def deposit(self, amount):
 if self._is_valid_amount(amount):
 self._balance += amount

 def _is_valid_amount(self, amount):
 return amount > 0
    ```

- The first argument for class methods should always be named cls in lowercase.
```
CORRECT
class Machine:
 count = 0

 def __init__(self, brand, model):
 self.brand = brand
 self.model = model
 Machine.count += 1

 @classmethod
 def get_count(cls):
 return cls.count
    ```

- Class attributes should not repeat class name:
```
WRONG
class Order:
 def __init__(self, order_id, order_state):
 self.order_id = order_id
 self.order_state = order_state
```
```
CORRECT
class Order:
 def __init__(self, id_, state):
 self.id_ = id_
 self.state = state
```

### Naming Object Variables
- Object variables should be written in lowercase snake_case.
- Object names should reflect the underlying class and may include adjectives or descriptive terms to clarify their role or function.
```
CORRECT
class Task:
 def __init__(self, name):
 self.name = name

build_task = Task("Build")
```

- For general concepts, including the class name in the object name may not be possible. Instead, the object's role should be clear from its name and context.
```
CORRECT
class Fruit:
 def __init__(self, name, color):
 self.name = name
 self.color = color

apple = Fruit("Apple", "Red")
```

```

```

banana = Fruit("Banana", "Yellow")
```

Naming Functions, method
- Function and method names should be written in lowercase snake_case.
- The first argument for methods should always be named self in lowercase.

4. Use Main Function and Main Name Idiom
- Always define a `main()` function to encapsulate the main logic of your script (put each step of the main logic into a separate function).
- Use the `if __name__ == "__main__":` idiom to ensure the script executes correctly when run directly, but not when imported.
```python
# WRONG
print("Hello World!")
```
```python
# BETTER
def main() -> None:
    print("Hello World!")

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

- Make sure the `main()` function is always the last function.
```python
# WRONG
def main() -> None:
    ...

def some_function() -> None:
    ...
```
```python
# CORRECT
def some_function() -> None:
    ...

def main() -> None:
    ...
```

- Each step in the `main()`-function should be encapsulated within a dedicated function
```python
# WRONG
def square(number):
    return number**2

def save_result(number, squared):
    with open("results.txt", "w") as file:
        file.write(f"The square of {number} is {squared}\n")
    print("Result saved to results.txt")

def main():
    while True:
        try:
            number = int(input("Enter a number to be squared: "))
            break
        except ValueError:
```

```

```

 print("Invalid input. Please enter a valid number (int).")
 squared = square(number)
 save_result(number, squared)
```
```python
CORRECT
def get_number():
 while True:
 try:
 return int(input("Enter a number to be squared: "))
 except ValueError:
 print("Invalid input. Please enter a valid number (int).")

def square(number):
 return number**2

def save_result(number, squared):
 with open("results.txt", "w") as file:
 file.write(f"The square of {number} is {squared}\n")
 print("Result saved to results.txt")

def main():
 number = get_number()
 squared = square(number)
 save_result(number, squared)
```

- Eventough the python community debates about, whether adding a docstring to the main() function is necessary, it increases clarity for unexperienced developers.
```python
BEST
def main() -> None:
 """Executes the main functionality of the script."""
 print("Hello World!")

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

- Never put the defintion of a class/function in the main() function. It should only demonstrate the main logic/functionality.
```python
WRONG
def main() -> None:
 """Executes the main functionality of the script."""
 class Machine:
 ...
```
```python
CORRECT
class Machine:
 ...
def main() -> None:
 """Executes the main functionality of the script."""
 machine = Machine(...)

```

## 5. Use Type Annotations/Hinting
- Keep logging as is (no type hints needed).
- Do not use Type Hints for global variables

```

```

- Use type hints to keep your code clean and intuitive.
```python
variable: type = value
```
```python
def function_name(param1: param1_type, param2: param2_type) -> return_type:
```

- Assign type hints to variables
```python
CORRECT - Python 3.10+
name: str = "Amadeus"
age: int = 66
success_rate: float = 0.95
machine_has_stopped: bool = False
payload: bytes = b'Sensor_Data'
last_name_to_age: dict[str, int] = {"Mozart": 66, "Beethoven": 99}
grades: list[int] = [1, 4, 2, 2, 3]
prime_numbers: set[int] = {2, 3, 5, 7}
screen_resolution: tuple[int, int] = (1920, 1080)
movie_ratings: list[float | str] = [1.5, "Great movie!", "N/A", 3.5]
username: str | None = get_username(user_id) if user_is_logged_in(user_id) else None
```
```python
CORRECT - Python < 3.10
from typing import Dict, List, Optional, Set, Tuple, Union

name: str = "Amadeus"
age: int = 66
success_rate: float = 0.95
machine_has_stopped: bool = False
payload: bytes = b'Sensor_Data'
last_name_to_age: Dict[str, int] = {"Mozart": 66, "Beethoven": 99}
grades: List[int] = [1]
prime_numbers: Set[int] = {6, 7}
screen_resolution: Tuple[int, int] = (1920, 1080)
movie_ratings: list[Union[float, str]] = [1.5, "Great movie!", "N/A", 3.5]
username: Optional[str] = get_username(user_id) if user_is_logged_in(user_id) else None
```

- Declare return_type
- return_type=None if function has no return statement.
```python
def log_error(error_message: str) -> None
 print(f"[ERROR] {error_message}")
```
- else define correct type
```python
Python 3.10+
def find_maximum(values: list[int]) -> int | None:
 if values:
 return max(values)
 else:
 return None
```
```python
Python <3.10
from typing import List, Optional

def find_maximum(values: List[int]) -> Optional[int]:
 if values:
 return max(values)
 else:
 return None
```

```

- Use inline comment "# type ignore" when importing third_party modules, that don't use type hinting

```
## 6. Import Modules Correctly
- Imports should be grouped into `Standard library imports`/`Third party imports`/`Local imports`, seperated with a blank line inbetween and use stand-alone comments above each category section. (if a category is not used, then the comment should be deleted)
```python
WRONG
import requests
from local_module import local_component
import time
from flask import Flask
import os
```
```python
CORRECT
Standard library imports
import time
import os

Third party imports
from flask import Flask
import requests

Local imports
from local_module import local_component
```

- Imports should be at the top of the file and never in a function to avoid performance losses.
```python
WRONG
import module

def function_name(...) -> None:
 import another_module
 ...
```
```python
CORRECT
import module
import another_module

def function_name(...) -> None:
 ...
```

- All imports should be sorted alphabetically and on seperate lines (top to bottom).
```python
WRONG
import module, another_module
```
```python
CORRECT
import another_module
import module
```

- When importing more than 3 components from the same module, use `import module`.
(Exception is module `typing`)
```python
CORRECT
```

```

```

import module
```
```python
# WRONG
from module import component_1, component_2, component_3, component_4
```

- When importing less than 4 components from the same module, use `from module import component`.
  ```python
  # CORRECT
  from module import component_1, component_2, component_3
  ```

  ```python
  # WRONG
  import module
  module.component_1()
  module.component_2()
  module.component_3()
  ```

- All imported components should be sorted alphabetically (left to right).
  ```python
  # WRONG
  from module
  ```

  ```python
  # CORRECT
  import another_module
  import module
  ```

- If necessary, use `import module as alias` to simplify long module names.
  ```python
  # CORRECT
  import very_very_long_module_name as alias
  ```

  ```python
  # WRONG
  import very_very_long_module_name
  ```

- Sometimes established Aliases can be used for commonly used modules.
  ```python
  # CORRECT
  import matplotlib.pyplot as plt
  import pandas as pd
  ```

- Sometimes Aliases are necessary if modules use the same naming on classes or functions.
  ```python
  # CORRECT
  from module_1 import component as component_1
  from module_2 import component as component_2
  ```

  ```python
  # WRONG
  from module_1 import component
  from module_2 import component
  ```

- Avoid wildcards imports `from module import *` as it makes the source of components unclear.
  ```python

```

```

# CORRECT
from module import component
```
```python
# WRONG
from module import *
```

- Imports should always be absolute and not relative
```
```python
CORRECT
from package_1.module_1 import component_1
```
```python
WRONG
from ..package_1.module_1 import component_1
```

- Remove imports, that are not used in the file
```
```python
# CORRECT
from module_2 import function_2

function_2()
```
```python
# WRONG
from module_1 import function_1
from module_2 import Class_2, function_2

function_2()
```

7. Docstrings
DOCSTRINGS must be done for CLASSES, FUNCTIONS/METHODS and MODULES!
Do not remove any comments! (but still do the module docstring)
General Docstring rules
- No blank line before the Docstring
```
```python
WRONG
def function_name(...) -> return_type:

"""
Docstring goes here.
"""

return return_value
```
```
```python
# CORRECT
def function_name(...) -> return_type:
"""
Docstring goes here.
"""

return return_value
```

- The indentation of the docstring should match the given code.
```
```python
WRONG
def function_name(...) -> return_type:
 """Docstring goes here.
 """

 return return_value
```

```

```

```
```python
# WRONG
def function_name(...) -> return_type:
    """
    Docstring goes here."""

    return return_value
```
```python
# CORRECT
def function_name(...) -> return_type:
    """
    Docstring goes here.

    """
    return return_value
```

- The docstring must be a complete sentence (ending with a period), describe the function's behavior as a command ("Do X & Return Y") and avoid passive descriptions ("Returns X").
```
```python
WRONG
def add_numbers(a: int, b: int) -> int:
 """
 Returns sum of two numbers
 """
 return a + b
```
```python
CORRECT
def add_numbers(a: int, b: int) -> int:
 """
 Add two numbers and return their sum.
 """
 return a + b
```

- The docstring should not contain redundant repetitions.
```
```python
# WRONG
def add_numbers(a: int, b: int) -> int:
    """
    add_numbers(a, b) -> int
    """
    return a + b
```
```python
# CORRECT
def add_numbers(a: int, b: int) -> int:
    """
    Add two numbers and return their sum.
    """
    return a + b
```

- If the docstring contains a backslash (\\"), use r"""\raw triple quotes"" to avoid formatting issues.
```
```python
WRONG
def read_file(file_path: str) -> str:
 """
 ...
```

```

```

Examples
-----
>>> read_file(r"C:\Users\Username\new_folder\file.txt")
Hello there!
"""
with open(file_path, 'r') as file:
    return file.read()
...
``python
# CORRECT
def read_file(file_path: str) -> str:
    """
    ...
    Examples
    -----
    >>> read_file(r"C:\Users\Username\new_folder\file.txt")
    Hello there!
    """
    with open(file_path, 'r') as file:
        return file.read()
```

Add Function/Method Docstrings by using the following rules
- No blank line after the function/method docstring
``python
WRONG
def function_name(...) -> return_type:
 """
 Docstring goes here.
 """
 return return_value
```
``python
# CORRECT
def function_name(...) -> return_type:
    """
    Docstring goes here.
    """
    return return_value
```

- For methods: Self must not be listed in the parameter section
``python
WRONG
class Calculator:
 def add(self, a: int, b: int) -> int:
 """
 ...
 Parameters

 self : Calculator
 The instance of the class.
 a : int
 First number.
 b : int
 Second number.

 ...
```

```

```

        return a + b
```
```python
# CORRECT
class Calculator:
    def add(self, a: int, b: int) -> int:
        """
        ...

        Parameters
        -----
        a : int
            First number.
        b : int
            Second number.

        ...
        """
        return a + b
```
- Function & method docstrings should always follow this pattern:
```python
def function_name(param1: param1_type, param2: param2_type) -> return_type:
    """
    Brief description of the function as an imperative command ("Do X & return Y")

    Detailed explanation of what the function does.
    Why is it needed? Are Error raised? If so, when?
    What steps or calculations does it perform?

    Parameters
    -----
    param1: param1_type
        Description of the first parameter.
    param2: param2_type
        Description of the second parameter.

    Returns
    -----
    return_type
        Description of what the function returns.

    Raises (if applicable)
    -----
    <ErrorType>
        Description of the exception raised if an error occurs.

    Examples
    -----
    >>> function_name(param1_value, param2_value)
    <expected_return_value_of_given_example>
    """
    return return_value
```
- This is an example, where the given pattern has been applied.
```python
def calculate_total_with_tax(prices: list[int | float], tax_rate: float) -> float:
    """
    Sums up given prices, applies the given tax rate, and returns the total price.

    This function sums up the provided price list and applies the given tax rate.
```

```

```

It raises a ValueError if the tax rate is negative or if any price is
either non-numeric or negative.

Parameters

prices: list[int | float]
 A list of item prices.
tax_rate: float
 The tax rate to apply.

Returns

float
 The total price after applying the tax.

Raises

ValueError
 If the tax rate is negative, or
 if any price is non-numeric or negative.

Examples

>>> calculate_total_with_tax([100.2, 50, 25.5], 0.1)
193.27
"""
if tax_rate < 0:
 raise ValueError(f"Invalid tax rate: {tax_rate}. Must be a non-negative
value.")

prices_are_non_numeric = not all(
 isinstance(price, (int, float)) for price in prices
)
if prices_are_non_numeric:
 raise ValueError("All prices must be of numeric (int/float)")

prices_are_negative = any(price < 0 for price in prices)
if prices_are_negative:
 raise ValueError("All prices must be non-negative.")

total: float = sum(prices)
total_with_tax = total * (1 + tax_rate)
return total_with_tax
```

### ## Add class docstrings with the following rules

- One blank line after the class docstring



```

```python
# WRONG
class SomeClass:
    """
    ...
    """
    def __init__(self, attribute1: attribute1_type) -> None:
        self.attribute1: attribute1_type = attribute1
```


```

```python
CORRECT
class SomeClass:
 """
 ...
 """

```


```


```

```

 def __init__(self, attribute1: attribute1_type) -> None:
 self.attribute1: attribute1_type = attribute1
 ...

 - The Methods section of a class-docstring should not list `self` as the first parameter,
 nor should it include any private methods.
        ```python
        # WRONG
        class SomeClass:
            """
            ...
            Methods
            -----
            method1(param1)
                Brief description of what method1 does.
            _private_method()
                This is a private method and should not be listed.

            ...
            """

        def __init__(self, attribute1: attribute1_type) -> None:
            self.attribute1: attribute1_type = attribute1

        def _private_method(self) -> None:
            pass

        ...
        ````python
 # CORRECT
 class SomeClass:
 """
 ...
 Methods

 method1(param1)
 Brief description of what method1 does.

 ...
 """

 def __init__(self, attribute1: attribute1_type) -> None:
 self.attribute1: attribute1_type = attribute1

 def _private_method(self) -> None:
 pass

 ...
        ````

        - Always document all attributes in class-docstrings, this includes those initialized via
        the constructor and those initialized internally with a default value.
        ```python
 # WRONG
 class SomeClass:
 """
 ...
 Attributes

```

```

attribute1: attribute1_type
 Description of the first attribute (initialized via the constructor).

"""

def __init__(self, attribute1: attribute1_type) -> None:
 self.attribute1: attribute1_type = attribute1
 self.attribute2: attribute1_type = False

 ...
```
```python
CORRECT
class SomeClass:
 """
 ...

 Attributes

 attribute1: attribute1_type
 Description of the first attribute (initialized via the constructor).
 attribute2: attribute2_type
 Description of the second attribute (initialized internally).
 ...
"""

 def __init__(self, attribute1: attribute1_type) -> None:
 self.attribute1: attribute1_type = attribute1
 self.attribute2: attribute1_type = False

 ...
```
- Class docstrings should always follow this pattern:
```
```python
class SomeClass:
    """
    Brief description of the class.

    Detailed explanation of what the class does.
    Why is it needed? Are Error raised? If so, when?
    What steps or calculations does it perform?

    Attributes
    -----
    attribute1: attribute1_type
        Description of the first attribute.
    attribute2: attribute2_type
        Description of the second attribute.

    Methods
    -----
    method1(param1)
        Description of what method1 does.
    """

    def __init__(self, attribute1: attribute1_type) -> None:
        self.attribute1: attribute1_type = attribute1
        self.attribute2: attribute1_type = False

    def method1(self, param1: param1_type) -> return_type:
        ...
```

```

```

def _private_method(self) -> None:
 ...
```
- This is an example, where the given pattern has been applied.
```python
class Machine:
 """
 A class used to represent a Machine.

 This class models a basic machine with functionality to start and stop.
 It keeps track of its running state.

 Attributes

 brand: str
 The brand of the machine.
 model: str
 The model of the machine.
 is_running: bool
 Indicates whether the machine is currently running.

 Methods

 start():
 Starts the machine, changing its state to running.
 stop():
 Stops the machine, changing its state to not running.
 """

def __init__(self, brand: str, model: str) -> None:
 """
 Constructs all the necessary attributes for the Machine object.

 Parameters

 brand: str
 The brand of the machine.
 model: str
 The model of the machine.
 """
 self.brand: str = brand
 self.model: str = model
 self.is_running: bool = False

def start(self) -> None:
 """
 Starts the machine.

 This method changes the state of the machine to running and outputs
 a message indicating that the machine has started.
 """
 self.is_running = True
 print(f"{self.brand} {self.model} started.")

def stop(self) -> None:
 """
 Stops the machine.

 This method changes the state of the machine to not running and
 outputs a message indicating that the machine has stopped.
 """
 self.is_running = False

```

```

 print(f"{self.brand} {self.model} stopped.")

```

```

Add module docstrings with the following rules

- Two blank lines after the module docstring

```

```python
WRONG
"""
Module name
=====
...
```

```

```

class SomeClass:
    ...
```

```

```

```python
# WRONG
"""
Module name
=====
...
```

```

```

class SomeClass:
 ...
```

```

CORRECT

```

```python
"""
Module name
=====
...
```

```

```

class SomeClass:
    ...
```

```

- Module docstrings should always follow this pattern:

```

```python
"""
Module name
=====
Brief description of the module.

Detailed explanation of the module (including its purpose and any other relevant
details)

```

Examples

Provide example usage patterns.

```

>>> from module_name import some_function
>>> result = some_function(arguments)
>>> print(result)

>>> from module_name import SomeClass
>>> some_object = SomeClass(arguments)
>>> some_object.some_method()
"""

```

```

def some_function():
    ...

class SomeClass:
    ...
    ...

- This is an example, where the given pattern has been applied.
```python
"""
Machine utils
=====

A simple module for representing and managing machines.

This module provides a class to model a basic machine with the ability to start and stop.
It keeps track of the machine's running state and provides methods to control it.

Examples

>>> from machine_utils import Machine
>>> machine_1 = Machine("Honda", "GX200")
>>> machine_1.start()
Honda GX200 started.

>>> machine_1.stop()
Honda GX200 stopped.
"""

class Machine:
 """
 ...
 ...

 def __init__(self, brand: str, model: str) -> None:
 ...
 ...
 self.brand: str = brand
 self.model: str = model
 self.is_running: bool = False

 def start(self) -> None:
 ...
 ...
 self.is_running = True
 print(f"{self.brand} {self.model} started.")

 def stop(self) -> None:
 ...
 ...
 self.is_running = False
 print(f"{self.brand} {self.model} stopped.")
 ...

8. Formatting & spacing

```

```
Make sure unused imports are removed

For strings use double quote strings ("), not single quotes(')
- Be careful, when changing f-strings to not impact the codes functionality

Make the code PEP-8 compliant

Limit the line length
- Limit long blocks of text (docstrings or comments) to maximum 72 characters.
- Limit all other lines to maximum 79 characters.
- Be careful to not accidentally remove any necessary characters (like f-strings)
```

## Anhang B: optipy.py

```
"""
OptiPy
=====

Optipy is a code optimization tool that uses various LLM providers to improve Python code quality.

This module provides functionality to optimize Python code according to specified guidelines
using different LLM providers (Anthropic, Google, OpenAI) and optimization strategies.

Examples

Command-line usage:

.. code-block:: bash
$ python optipy.py --filepath=example.py --strategy=one_by_one --model=gpt-4o --debug

Library usage:

.. code-block:: python
 from pathlib import Path
 from optipy import Optipy

 optipy = Optipy(
 filepath=Path("example.py"),
 strategy="one_by_one",
 model="gpt-4o",
 debug=True
)

 optipy.run_optimization()
 # print(optipy.get_original_code())
 # print(optipy.get_optimized_code())
"""

Standard library imports
import argparse
import os
from pathlib import Path
import re
from typing import get_args, Literal, TypeAlias, TypedDict
import time

Third party imports
import anthropic
import autopep8 # type: ignore
from dotenv import load_dotenv
from google import genai # type: ignore
from google.genai import types # type: ignore
from openai import OpenAI

Local imports
import toolbox

OptimizationStrategy: TypeAlias = Literal["all_at_once", "one_by_one"]

class LLMResponse(TypedDict):
 """
```

```

A TypedDict representing a response from a Language Learning Model (LLM).

This structure encapsulates the content returned by the LLM along with token usage
metrics
for both input and output. It provides a standardized format for responses
across different LLM providers.

Attributes

content : str
 The text content returned by the LLM.
input_tokens : int
 The number of tokens used in the input prompt.
output_tokens : int
 The number of tokens generated in the output response.
"""

content: str
input_tokens: int
output_tokens: int

class OptipyConfig:
"""
Configuration class for Optipy to manage and validate optimization settings.

This class handles the selection of the LLM provider, model, API key retrieval,
and provides a structured way to access available models. It ensures the chosen
model belongs to a recognized provider and retrieves the appropriate API key
from environment variables.

Parameters

filepath : Path
 The path to the Python file to be optimized.
strategy : OptimizationStrategy
 The strategy for optimization ('one_by_one' or 'all_at_once').
model : str | None, optional
 The specific LLM model to use. Defaults to OpenAI's "gpt-4o-2024-08-06" if None is
provided.
debug : bool, optional
 Whether to enable debug mode for logging outputs, by default False.

Attributes

filepath : Path
 Stores the file path of the Python script to be optimized.
strategy : OptimizationStrategy
 Optimization strategy to use.
model : str
 Selected model for the optimization.
provider : str
 The LLM provider associated with the selected model (Anthropic, Google, OpenAI, or
XAI).
api_key : str
 API key retrieved from environment variables for the selected provider.

Methods

get_api_key() -> str
 Returns the API key for the configured provider.
get_available_models() -> dict[str, list[str]]
 Returns a dictionary of all available models grouped by provider.

```

```

"""
PROVIDERS = {
 "anthropic": [
 "claude-3-7-sonnet-20250219",
 "claude-3-5-sonnet-20241022",
 "claude-3-5-haiku-20241022",
 "claude-3-5-sonnet-20240620",
 "claude-3-haiku-20240307",
 "claude-3-opus-20240229",
],
 "google": [
 "chat-bison-001",
 "text-bison-001",
 "embedding-gecko-001",
 "gemini-1.0-pro-vision-latest",
 "gemini-pro-vision",
 "gemini-1.5-pro-latest",
 "gemini-1.5-pro-001",
 "gemini-1.5-pro-002",
 "gemini-1.5-pro",
 "gemini-1.5-flash-latest",
 "gemini-1.5-flash-001",
 "gemini-1.5-flash-001-tuning",
 "gemini-1.5-flash",
 "gemini-1.5-flash-002",
 "gemini-1.5-flash-8b",
 "gemini-1.5-flash-8b-001",
 "gemini-1.5-flash-8b-latest",
 "gemini-1.5-flash-8b-exp-0827",
 "gemini-1.5-flash-8b-exp-0924",
 "gemini-2.0-flash-exp",
 "gemini-2.0-flash",
 "gemini-2.0-flash-001",
 "gemini-2.0-flash-lite-001",
 "gemini-2.0-flash-lite",
 "gemini-2.0-flash-lite-preview-02-05",
 "gemini-2.0-flash-lite-preview",
 "gemini-2.0-pro-exp",
 "gemini-2.0-pro-exp-02-05",
 "gemini-exp-1206",
 "gemini-2.0-flash-thinking-exp-01-21",
 "gemini-2.0-flash-thinking-exp",
 "gemini-2.0-flash-thinking-exp-1219",
 "learnlm-1.5-pro-experimental",
 "embedding-001",
 "text-embedding-004",
 "gemini-embedding-exp-03-07",
 "gemini-embedding-exp",
 "aqa",
 "imagen-3.0-generate-002",
],
 "openai": [
 "gpt-4.5-preview",
 "gpt-4.5-preview-2025-02-27",
 "gpt-4o-mini-audio-preview-2024-12-17",
 "dall-e-3",
 "dall-e-2",
 "gpt-4o-audio-preview-2024-10-01",
 "gpt-4o-audio-preview",
 "gpt-4o-mini-realtime-preview-2024-12-17",
 "gpt-4o-mini-realtime-preview",
 "o1-mini-2024-09-12",
],
}

```

```

 "o1-mini",
 "omni-moderation-latest",
 "gpt-4o-mini-audio-preview",
 "omni-moderation-2024-09-26",
 "whisper-1",
 "gpt-4o-realtime-preview-2024-10-01",
 "babbage-002",
 "gpt-4-turbo-preview",
 "chatgpt-4o-latest",
 "tts-1-hd-1106",
 "text-embedding-3-large",
 "gpt-4-0125-preview",
 "gpt-4o-audio-preview-2024-12-17",
 "gpt-4",
 "gpt-4o-2024-05-13",
 "tts-1-hd",
 "o1-preview",
 "o1-preview-2024-09-12",
 "gpt-4o-2024-11-20",
 "gpt-3.5-turbo-instruct-0914",
 "gpt-4o-mini-2024-07-18",
 "gpt-4o-mini",
 "tts-1",
 "tts-1-1106",
 "davinci-002",
 "gpt-3.5-turbo-1106",
 "gpt-4-turbo",
 "gpt-3.5-turbo-instruct",
 "o1",
 "gpt-4o-2024-08-06",
 "gpt-3.5-turbo-0125",
 "gpt-4o-realtime-preview-2024-12-17",
 "gpt-3.5-turbo",
 "gpt-4-turbo-2024-04-09",
 "gpt-4o-realtime-preview",
 "gpt-3.5-turbo-16k",
 "gpt-4o",
 "text-embedding-3-small",
 "gpt-4-1106-preview",
 "text-embedding-ada-002",
 "o3-mini-2025-01-31",
 "gpt-4-0613",
 "o3-mini",
 "o1-2024-12-17",
],
 "xai": [
 "grok-2-1212",
 "grok-2-vision-1212",
 "grok-beta",
 "grok-vision-beta",
],
}
}

DEFAULT_MODELS = {
 "anthropic": "claude-3-7-sonnet-20250219",
 "google": "gemini-2.0-pro-exp-02-05",
 "openai": "gpt-4o-2024-08-06",
 "xai": "grok-2-1212",
}
}

def __init__(
 self,
 filepath: Path,

```

```

 strategy: OptimizationStrategy,
 model: str | None = None,
 debug: bool = False,
) -> None:
 """Initializes the configuration for Optipy."""
 self.filepath = filepath
 self.strategy = strategy
 self.model = model or self.DEFAULT_MODELS["openai"]
 self.debug = debug
 self.provider = self._get_provider()
 self.api_key = self._read_env()

 def _get_provider(self) -> str:
 """Determines the provider based on the selected model."""
 for provider, models in self.PROVIDERS.items():
 if self.model in models:
 return provider
 available_models = [
 model for models in self.PROVIDERS.values() for model in models
]
 raise ValueError(
 f"Unknown model: {self.model}. Available models: {',
'.join(sorted(available_models))}"
)

 def _read_env(self) -> str:
 """Loads and retrieves the API key from the environment."""
 load_dotenv(override=True)
 env_vars = {
 "anthropic": "ANTHROPIC_API_KEY",
 "google": "GOOGLE_API_KEY",
 "openai": "OPENAI_API_KEY",
 "xai": "XAI_API_KEY",
 }
 api_key = os.getenv(env_vars[self.provider])
 if api_key is None:
 raise ValueError(f"{env_vars[self.provider]} environment variable not set")
 return api_key

 def get_api_key(self) -> str:
 """Returns the API key for the configured provider."""
 return self.api_key

 def get_available_models(self) -> dict[str, list[str]]:
 """Returns a dictionary of all available models grouped by provider."""
 return self.PROVIDERS

class Optipy:
 """
 A class to optimize Python code based on predefined guidelines using different LLM providers.

 Parameters

 filepath : Path
 The path to the Python code file that needs optimization.
 strategy : OptimizationStrategy
 The optimization strategy to use ('one_by_one' or 'all_at_once').
 model : str | None, optional
 The LLM model to use for optimization, by default None.
 debug : bool, optional
 Whether to enable debug mode for logging outputs, by default False.
 """

```

```

Attributes

config : OptipyConfig
 Configuration settings for the optimization process.
guideline : str
 The content of the guideline file used for optimization.
guideline_rules : list[str]
 The extracted rules from the guideline.
guideline_titles : list[str]
 The extracted titles of the guideline rules.
original_code : str
 The original Python code before optimization.
optimized_code : str | None
 The optimized Python code after processing, or None if not optimized yet.

Methods

run_optimization()
 Runs the optimization process using the selected strategy.
get_original_code() -> str
 Returns the original Python code before optimization.
get_optimized_code() -> str
 Returns the optimized Python code after processing.
"""

def __init__(
 self,
 filepath: Path,
 strategy: OptimizationStrategy,
 model: str | None = None,
 debug: bool = False,
) -> None:
 """Initializes Optipy with configuration settings."""
 self.config = OptipyConfig(filepath, strategy, model, debug)
 self.guideline = self._read_guideline()
 self.guideline_rules = self._extract_guideline_rules()
 self.guideline_titles = self._extract_guideline_titles()
 self.original_code = self._read_input_code()
 self.optimized_code: str | None = None

def _get_llm_response(self, original_code: str, prompt: str) -> LLMResponse:
 """Fetches response from the selected LLM provider."""
 if self.config.provider == "anthropic":
 return self._get_anthropic_response(original_code, prompt)
 if self.config.provider == "google":
 return self._get_google_response(original_code, prompt)
 if self.config.provider == "openai":
 return self._get_openai_response(original_code, prompt)
 if self.config.provider == "xai":
 return self._get_xai_response(original_code, prompt)
 raise ValueError(f"Unsupported provider: {self.config.provider}")

def _get_anthropic_response(self, original_code: str, prompt: str) -> LLMResponse:
 """Gets a response from Anthropic provider."""
 client = anthropic.Anthropic(api_key=self.config.api_key)
 message = client.messages.create(
 model=self.config.model,
 temperature=0,
 max_tokens=8192,
 system=prompt,
 messages=[
 {

```

```

 "role": "user",
 "content": [{"type": "text", "text": original_code}],
],
)
)

if not hasattr(message.content[0], "text"):
 raise ValueError("Invalid response from Anthropic API")

response: LLMResponse = {
 "content": message.content[0].text,
 "input_tokens": int(message.usage.input_tokens),
 "output_tokens": int(message.usage.output_tokens),
}
return response

def _get_google_response(self, original_code: str, prompt: str) -> LLMResponse:
 """Gets a response from Google provider."""
 client = genai.Client(api_key=self.config.api_key)
 response = client.models.generate_content(
 model=self.config.model,
 contents=original_code,
 config=types.GenerateContentConfig(
 system_instruction=prompt, temperature=0
),
)
 response_data: LLMResponse = {
 "content": response.text,
 "input_tokens": int(response.usage_metadata.prompt_token_count),
 "output_tokens": int(response.usage_metadata.candidates_token_count),
 }
 return response_data

def _get_openai_response(self, original_code: str, prompt: str) -> LLMResponse:
 """Gets a response from OpenAI provider."""
 client = OpenAI(api_key=self.config.api_key)
 chat_completion = client.chat.completions.create(
 model=self.config.model,
 messages=[
 {"role": "system", "content": prompt},
 {"role": "user", "content": original_code},
],
 temperature=0,
)

 if not chat_completion.usage or not chat_completion.choices[0].message.content:
 raise ValueError("Invalid response from OpenAI API")

 response_data: LLMResponse = {
 "content": chat_completion.choices[0].message.content,
 "input_tokens": chat_completion.usage.prompt_tokens,
 "output_tokens": chat_completion.usage.completion_tokens,
 }
 return response_data

def _get_xai_response(self, original_code: str, prompt: str) -> LLMResponse:
 """Gets a response from XAI provider."""
 client = OpenAI(
 api_key=self.config.api_key,
 base_url="https://api.x.ai/v1",
)
 chat_completion = client.chat.completions.create(
 model=self.config.model,

```

```

 messages=[
 {"role": "system", "content": prompt},
 {"role": "user", "content": original_code},
],
 temperature=0,
)

 if not chat_completion.usage or not chat_completion.choices[0].message.content:
 raise ValueError("Invalid response from XAI API")

 response_data: LLMResponse = {
 "content": chat_completion.choices[0].message.content,
 "input_tokens": chat_completion.usage.prompt_tokens,
 "output_tokens": chat_completion.usage.completion_tokens,
 }
 return response_data

def _read_guideline(self) -> str:
 """Reads the guideline file for optimization rules."""
 with Path("guideline.md").open("r", encoding="utf-8") as f_in:
 return f_in.read()

def _extract_guideline_rules(self) -> list[str]:
 """Extracts individual rules from the guideline."""
 return self.guideline.split("\#\# ")[1:]

def _extract_guideline_titles(self) -> list[str]:
 """Extracts titles of guideline rules."""
 pattern: str = r"^\#\#\s+(?:\d+\.\s*)?(.+)"
 return re.findall(pattern, self.guideline, re.MULTILINE)

def _read_input_code(self) -> str:
 """Reads the input Python file for optimization."""
 with self.config.filepath.open("r", encoding="utf-8") as f_in:
 return f_in.read()

def _create_optimization_prompt(self, rules: str | list[str]) -> str:
 """Creates a prompt for the LLM optimization request."""
 rules_text: str
 if isinstance(rules, list):
 joined_rules: str = "\n## ".join(rules)
 rules_text = f"rules: \n## {joined_rules}"
 else:
 rules_text = f"rule: \n## {rules}"

 optimization_prompt: str = f"""# Role
- You are an Expert in establishing code quality in Python (3.10+).

Context
- Give back the code in markdown format.
- Do not include any explanation.
- Modify only what is mentioned in the provided guidelines.
- If the code is not compliant with the given guideline, you will be fined $1000!

Instructions
- Your task is to ensure that the given code adheres to the following
{rules_text}
"""

 return optimization_prompt

def _extract_optimized_code(self, llm_response: str) -> str:
 """Extracts the optimized code from the LLM response."""
 if not llm_response:

```

```

 raise ValueError("No response received from the LLM.")
 pattern: str = r"```python\n(.*?)\n```"
 matches = re.findall(pattern, llm_response, re.DOTALL)
 if not matches:
 print(str(llm_response))
 raise ValueError(
 "No Python code block found in LLM response, check max_tokens/rate_limits!"
)
 return matches[0]

def _format_code(self, code: str) -> str:
 """Formats the code using autopep8."""
 return autopep8.fix_code(code)

def _save_optimized_code(self, optimized_code: str) -> None:
 """Saves the optimized code to a new file."""
 suffix: str = (
 "_optimized_aao"
 if self.config.strategy == "all_at_once"
 else "_optimized_obo"
)
 optimized_filepath = self.config.filepath.with_stem(
 self.config.filepath.stem + suffix
)
 with optimized_filepath.open("w", encoding="utf-8") as f_out:
 f_out.write(optimized_code)

def _optimize_one_by_one(self) -> str:
 """Optimizes code one rule at a time."""
 optimized_code = self.original_code
 for i, rule in enumerate(self.guideline_rules):
 start_time = time.time()
 prompt = self._create_optimization_prompt(rule)
 llm_response = self._get_llm_response(optimized_code, prompt)
 llm_content = llm_response["content"]
 used_input_tokens = llm_response["input_tokens"]
 used_output_tokens = llm_response["output_tokens"]
 optimized_code = self._extract_optimized_code(llm_content)
 execution_time = time.time() - start_time
 if self.config.model == "gemini-2.0-pro-exp-02-05" and execution_time < 12:
 time.sleep(12 - execution_time)
 execution_time = time.time() - start_time

 if self.config.debug:
 print(optimized_code)

 print(
 f"✓ [{i+1}/{len(self.guideline_titles)}]"
 f"[{self.guideline_titles[i]}] Successfully applied rule in "
 f"{execution_time:.2f}s"
 f"({used_input_tokens=} | {used_output_tokens=})"
)

 return optimized_code

def _optimize_all_at_once(self) -> str:
 """Optimizes code by applying all rules at once."""
 start_time = time.time()
 prompt = self._create_optimization_prompt(self.guideline_rules)
 llm_response = self._get_llm_response(self.original_code, prompt)
 llm_content = llm_response["content"]
 used_input_tokens = llm_response["input_tokens"]
 used_output_tokens = llm_response["output_tokens"]

```

```

 optimized_code = self._extract_optimized_code(llm_content)
 execution_time = time.time() - start_time

 if self.config.debug:
 print(optimized_code)

 print(
 f"✓ Applied all {len(self.guideline_titles)} rules at once in "
 f"{execution_time:.2f}s ({used_input_tokens=} | {used_output_tokens=})"
)

 return optimized_code

@toolbox.measure_exec_time
def run_optimization(self) -> None:
 """Run the optimization process using the selected strategy."""
 if self.config.strategy == "one_by_one":
 optimized_code = self._optimize_one_by_one()
 else:
 optimized_code = self._optimize_all_at_once()

 formatted_code = self._format_code(optimized_code)
 if self.config.debug:
 print(formatted_code)
 self.optimized_code = formatted_code
 self._save_optimized_code(formatted_code)

def get_original_code(self) -> str:
 """Returns the original code before optimization."""
 return self.original_code

def get_optimized_code(self) -> str:
 """Returns the optimized code after processing."""
 if self.optimized_code is None:
 raise ValueError("No optimized code available. Run optimize() first.")
 return self.optimized_code

def get_optipy_args() -> argparse.Namespace:
 """Parse command line arguments for file path, strategy, model, and debug options."""
 parser = argparse.ArgumentParser(
 description="Optimize code quality from a given file."
)
 parser.add_argument(
 "--filepath", type=Path, required=True, help="Path to the input file"
)
 parser.add_argument(
 "--strategy",
 type=str,
 choices=get_args(OptimizationStrategy),
 default="one_by_one",
 help="Optimization strategy to use (default: one_by_one)",
)
 parser.add_argument(
 "--model",
 type=str,
 help="Model to use (default: gpt-4o from OpenAI)",
)
 parser.add_argument(
 "--debug",
 action="store_true",
 help="Enable debug mode to print optimized code",
)

```

```
return parser.parse_args()

def main() -> None:
 """Executes the main functionality of the script."""
 args = get_optipy_args()

 optipy = Optipy(
 filepath=args.filepath,
 strategy=args.strategy,
 model=args.model,
 debug=args.debug,
)
 optipy.run_optimization()
 # print(optipy.get_original_code())
 # print(optipy.get_optimized_code())

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

## Anhang C: metrics\_analyzer.py

```
"""
Metrics Analyzer
=====

Module for evaluating and analyzing Python file metrics including
complexity and maintainability.

This module provides tools to analyze Python code for various quality
metrics.

Examples

Command-line usage:

.. code-block:: bash
$ python metrics_analyzer.py --filepath=example.py

Library usage:

.. code-block:: python
 from metrics_analyzer import MetricsAnalyzer
 analyzer = MetricsAnalyzer("example.py")
 metrics = analyzer.get_metrics()
 analyzer.display_metrics()

Standard library imports
import argparse
import ast
import math
import re
import subprocess

Third party imports
import pandas as pd
from radon.complexity import cc_visit
from radon.metrics import h_visit
from radon.raw import analyze

class MetricsAnalyzer:
 """Class for analyzing Python file metrics including complexity and maintainability."""

 def __init__(self, file_path):
 """Initialize the analyzer with a file path.

 Args:
 file_path: Path to the Python file to analyze
 """
 self.file_path = file_path
 self.code = None
 self.tree = None
 self.metrics = None

 def read_file(self):
 """Read the file content."""
 try:
 with open(self.file_path, "r", encoding="utf-8") as file_handle:
 self.code = file_handle.read()
 return True
 except (IOError, FileNotFoundError, PermissionError) as exc:
```

```

 print(f"Error reading file: {exc}")
 return False

 def get_ast_tree(self):
 """Parse the file and return the AST."""
 if not self.code:
 return False
 try:
 self.tree = ast.parse(self.code, filename=self.file_path)
 return True
 except SyntaxError:
 print(f"Syntax error in {self.file_path}")
 return False

 def get_function_length(self):
 """Calculate the average function length."""
 if not self.tree:
 return 0

 function_lengths = []
 for node in ast.walk(self.tree):
 if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
 start_line = node.lineno
 end_line = max(n.lineno for n in ast.walk(node) if hasattr(n, "lineno"))
 function_lengths.append(end_line - start_line + 1)

 return sum(function_lengths) / len(function_lengths) if function_lengths else 0

 def get_function_length_without_docstrings(self):
 """Calculate the average function length without counting docstring lines."""
 if not self.tree:
 return 0

 function_lengths = []
 for node in ast.walk(self.tree):
 if isinstance(node, (ast.FunctionDef, ast.AsyncFunctionDef)):
 start_line = node.lineno
 end_line = max(n.lineno for n in ast.walk(node) if hasattr(n, "lineno"))
 total_length = end_line - start_line + 1

 docstring = ast.get_docstring(node)
 docstring_lines = 0

 if (
 docstring
 and node.body
 and isinstance(node.body[0], ast.Expr)
 and isinstance(node.body[0].value, ast.Str)
):
 docstring_node = node.body[0]

 if hasattr(docstring_node, "end_lineno"):
 docstring_lines = (
 docstring_node.end_lineno - docstring_node.lineno + 1
)
 else:
 docstring_text = docstring
 docstring_lines = (
 docstring_text.count("\n") + 2
 if "\n" in docstring_text
 else 1
)


```

```

adjusted_length = total_length - docstring_lines
function_lengths.append(adjusted_length)

return sum(function_lengths) / len(function_lengths) if function_lengths else 0

def get_docstring_ratio(self):
 """Calculate the percentage of documented constructs."""
 if not self.tree:
 return 0

 total_constructs = 0
 documented_constructs = 0
 for node in ast.walk(self.tree):
 if isinstance(
 node, (ast.FunctionDef, ast.AsyncFunctionDef, ast.ClassDef, ast.Module)
):
 total_constructs += 1
 if ast.get_docstring(node):
 documented_constructs += 1

 return (
 (documented_constructs / total_constructs) * 100
 if total_constructs > 0
 else 0
)

def get_maintainability_index(self, halstead_volume, avg_cc, sloc, docstring_ratio):
 """Calculate the maintainability index."""
 mi_formula = (
 171
 - 5.2 * math.log(halstead_volume)
 - (0.23 * avg_cc)
 - (16.2 * math.log(sloc))
 + (50 * (math.sin(math.sqrt(2.4 * math.radians(docstring_ratio)))))
)
 maintainability_index = 100 * mi_formula / 171
 return max(0, min(100, maintainability_index))

def rank_maintainability_index(self, mi_score):
 """Assign a ranking letter based on the Maintainability Index (MI) score."""
 if mi_score >= 20:
 return "A" # Very High
 if 10 <= mi_score < 20:
 return "B" # Medium
 return "C" # Extremely Low

def rank_cyclomatic_complexity(self, cc_score):
 """Assign a ranking letter based on the Cyclomatic Complexity (CC) score."""
 if cc_score <= 5:
 return "A" # Low - Simple block
 if 6 <= cc_score <= 10:
 return "B" # Low - Well structured and stable block
 if 11 <= cc_score <= 20:
 return "C" # Moderate - Slightly complex block
 if 21 <= cc_score <= 30:
 return "D" # More than moderate - More complex block
 if 31 <= cc_score <= 40:
 return "E" # High - Complex block, alarming
 return "F" # Very high - Unstable block, error-prone

def get pylint_score(self):
 """Run pylint and get the score."""
 try:

```

```

 result = subprocess.run(
 ["pylint", "--disable=W1203,R1705", self.file_path],
 capture_output=True,
 text=True,
 check=False,
)
 match = re.search(r"Your code has been rated at ([\d.]+)/10", result.stdout)
 return float(match.group(1)) if match else None
 except (subprocess.SubprocessError, ValueError, TypeError) as exc:
 print(f"Error running pylint: {exc}")
 return None

 def _extract_code_metrics(self):
 """Extract all code metrics from the given code and AST tree."""
 raw_metrics = analyze(self.code)

 complexity_results = cc_visit(self.code)
 halstead_results = h_visit(self.code)

 if complexity_results:
 total_complexity = sum(block.complexity for block in complexity_results)
 avg_cc = total_complexity / len(complexity_results)
 else:
 avg_cc = 0

 halstead_volume = halstead_results.total.volume

 avg_func_len = self.get_function_length()
 avg_func_len_no_docs = self.get_function_length_without_docstrings()
 docstring_ratio = self.get_docstring_ratio()
 pylint_score = self.get_pylint_score()
 maint_index = self.get_maintainability_index(
 halstead_volume, avg_cc, raw_metrics.sloc, docstring_ratio
)

 return {
 "file_path": self.file_path,
 "loc": raw_metrics.loc,
 "sloc": raw_metrics.sloc,
 "single_comments": raw_metrics.single_comments,
 "multi": raw_metrics.multi,
 "blank_lines": raw_metrics.blank,
 "comments": raw_metrics.comments,
 "avg_func_len": avg_func_len,
 "avg_func_len_no_docs": avg_func_len_no_docs,
 "avg_cc": avg_cc,
 "mi": maint_index,
 "pylint_score": pylint_score,
 }

 def get_metrics(self):
 """Analyze the file and return all metrics."""
 if not self.read_file():
 print(f"Cannot analyze empty file: {self.file_path}")
 return None

 if not self.get_ast_tree():
 print(f"Cannot parse file: {self.file_path}")
 return None

 self.metrics = self._extract_code_metrics()
 return self.metrics

```

```

def _calculate_percentage(self, value, total):
 """Calculate percentage of a value against a total."""
 return (value / total) * 100 if total > 0 else 0

def _prepare_metrics_for_display(self):
 """Prepare metrics data for display by calculating ratios and formatting values."""
 if not self.metrics:
 return {}

 metrics = self.metrics
 loc = metrics["loc"]

 ratios = {
 "sloc": self._calculate_percentage(metrics["sloc"], loc),
 "single_comments": self._calculate_percentage(
 metrics["single_comments"], loc
),
 "multi": self._calculate_percentage(metrics["multi"], loc),
 "blank_lines": self._calculate_percentage(metrics["blank_lines"], loc),
 "comments": self._calculate_percentage(metrics["comments"], loc),
 }

 total_percentage = round(
 ratios["sloc"]
 + ratios["single_comments"]
 + ratios["multi"]
 + ratios["blank_lines"],
 2,
)

 rounded = {
 "func_len": round(metrics["avg_func_len"], 2),
 "func_len_no_docs": round(metrics["avg_func_len_no_docs"], 2),
 "cc": round(metrics["avg_cc"], 2),
 "mi": round(metrics["mi"], 2),
 "pylint": (
 round(metrics["pylint_score"], 2)
 if metrics["pylint_score"] is not None
 else None
),
 }

 mi_rank = self.rank_maintainability_index(rounded["mi"])
 cc_rank = self.rank_cyclomatic_complexity(rounded["cc"])

 return {
 "LOC": [f"{loc} ({total_percentage}%)"],
 "SLOC": [f"{metrics['sloc']} ({ratios['sloc']:.2f}%)"],
 "SLCOM": [
 f"{metrics['single_comments']} ({ratios['single_comments']:.2f}%)"
],
 "MULTI": [f"{metrics['multi']} ({ratios['multi']:.2f}%)"],
 "BLANK": [f"{metrics['blank_lines']} ({ratios['blank_lines']:.2f}%)"],
 "/": ["/"],
 "Commented Lines": [f"{metrics['comments']} ({ratios['comments']:.2f}%)"],
 "Avg. func len (with docs)": [f"{rounded['func_len']:.2f}"],
 "Avg. func len (without docs)": [f"{rounded['func_len_no_docs']:.2f}"],
 "Avg. CC": [f"{rounded['cc']:.2f} ({cc_rank})"],
 "MI": [f"{rounded['mi']:.2f} ({mi_rank})"],
 "Pylint Score": [
 f"{rounded['pylint']:.2f}" if rounded["pylint"] is not None else "N/A"
],
 }
}

```

```

def display_metrics(self):
 """Format and display metrics in a table."""
 if not self.metrics:
 print("No metrics to display. Run analyze() first.")
 return None

 display_data = self._prepare_metrics_for_display()
 if not display_data:
 print("No metrics to display.")
 return None

 metrics_df = pd.DataFrame(display_data, index=[self.metrics["file_path"]])
 df_string = metrics_df.to_string(justify="center", col_space=12)
 separator = "=" * len(max(df_string.split("\n"), key=len))
 print(f"{separator}\n{df_string}\n{separator}")
 return df_string

def parse_arguments():
 """Parse command line arguments."""
 parser = argparse.ArgumentParser(description="Analyze Python file metrics")
 parser.add_argument("--filepath", type=str, help="Path to Python file")
 return parser.parse_args()

def main():
 """Execute the main functionality of the script."""
 args = parse_arguments()
 analyzer = MetricsAnalyzer(args.filepath)
 analyzer.get_metrics()
 analyzer.display_metrics()

if __name__ == "__main__":
 main()

```

## Anhang D: optipy-workflow.yml

```
name: Optipy

on:
 push:
 branches: [main]
 pull_request:
 branches: [main]

permissions:
 contents: write

jobs:
 optipy:
 runs-on: ubuntu-latest
 env:
 PYTHONDONTWRITEBYTECODE: 1

 steps:
 - uses: actions/checkout@v3
 with:
 fetch-depth: 0

 - name: Set up Python
 uses: actions/setup-python@v3
 with:
 python-version: '3.10'

 - name: Install dependencies
 run: |
 python -m pip install --upgrade pip
 pip install -r requirements.txt

 - name: Run optipy
 env:
 ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }}
 GOOGLE_API_KEY: ${{ secrets.GOOGLE_API_KEY }}
 OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
 XAI_API_KEY: ${{ secrets.XAI_API_KEY }}
 run: |
 DISABLE_OPTIMIZATION=true

 if ["$DISABLE_OPTIMIZATION" = true]; then
 echo "Optimization process is disabled"
 exit 0
 fi

 EXCLUDE_PATTERNS=(
 "./optipy.py"
 "./toolbox.py"
 "./case_studies/*"
)

 for file in $(find . -name "*.py"); do
 SKIP=false
 for pattern in "${EXCLUDE_PATTERNS[@]}"; do
 if [["$file" == $pattern]]; then
 echo "Skipping $file (matched pattern $pattern)"
 SKIP=true
 break
 fi
 done
 if ! $SKIP; then
 python $file
 fi
 done

```

```
done

if ["$SKIP" = false]; then
 echo "Processing $file"
 python optipy.py --filepath="$file" --model=gpt-4o --strategy=one_by_one --
debug
 fi
done

- name: Commit changes
 run: |
 git config --global user.name 'GitHub Actions'
 git config --global user.email 'actions@github.com'
 git add .
 if git diff --staged --quiet; then
 echo "No changes to commit"
 else
 git commit -m "Apply optipy changes"
 git push
 fi
```

## Anhang E: case\_study\_1.py

```
#initialize variables
a = 0
b = 0
res = 0

import random as rand

print('Simple Calculator')
print('Operations: +,-,*,/')

#get inputs
a = float(input("Enter first number that will be used as the first operand in the
mathematical operation that you want to perform: ")) #first number
b = float(input('Enter second number that will be used as the second operand in the
mathematical operation that you want to perform: ')) #second number
OP = input("Enter the desired mathematical operation that you would like to perform on the
two numbers you just provided (+, -, *, /): ") #what operation to do

#do the calculation
if OP=="+": #addition
 res=a+b
elif OP=="-": #subtraction
 res=a-b
elif OP=="*": #multiplication
 res=a*b
elif OP=="/": #division
 if b=="0":
 print('Cannot divide by zero') # error message
 else:
 res=a/b
else:
 print('Invalid operation') # invalid operation message

print("Result:",res)
print('Thank you for using the calculator!')
```

## Anhang F: case\_study\_2.py

```
import os, json, pyfiglet

data = []

def saveToFile(data_to_save):
 f = open("users.json", "w")
 f.write(json.dumps(data_to_save, indent=4))
 f.close()

def addUser(name, age, email, status):
 # adding a user with all details
 user = {}
 user["name"] = name
 user["age"] = age
 user["email"] = email
 user["status"] = status
 data.append(user)
 saveToFile(data)
 print("User added: " + name)

def updateUser(name, field, value):
 # update user info based on name
 found = 0
 for i in range(len(data)):
 if data[i]["name"] == name:
 data[i][field] = value
 found = 1
 if found == 0:
 print("User not found")
 else:
 saveToFile(data)
 print("Updated " + name + " " + field)

def getUser(name):
 # find and return user
 for i in range(len(data)):
 if data[i]["name"] == name:
 return data[i]
 return "Not found"

def main():
 # main program logic
 header = pyfiglet.figlet_format("User Manager")
 print(header)
 while True:
 print("1. Add user")
 print("2. Update user")
 print("3. Get user")
 print("4. Exit")
 choice = input("Enter choice: ")

 if choice == "1":
 name = input("Enter name: ")
 age = input("Enter age: ")
 email = input("Enter email: ")
 status = input("Enter status (active/inactive): ")
 addUser(name, age, email, status)
 elif choice == "2":
 name = input("Enter name to update: ")
 field = input("Enter field to update (name/age/email/status): ")
```

```
 value = input("Enter new value: ")
 updateUser(name, field, value)
 elif choice == "3":
 name = input("Enter name to find: ")
 result = getUser(name)
 if result == "Not found":
 print("User not found")
 else:
 print("Name: " + result["name"])
 print("Age: " + result["age"])
 print("Email: " + result["email"])
 print("Status: " + result["status"])
 elif choice == "4":
 print("Exiting...")
 break
 else:
 print("Invalid choice, try again")

Check if file exists and load it
if os.path.exists("output.json"):
 f = open("output.json", "r")
 data.clear()
 data.extend(json.load(f))
 f.close()

Start the program
main()
```

## Anhang G: case\_study\_3.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import solve_ivp # type: ignore
import math
from termcolor import colored
import colorama
from dataclasses import dataclass
from typing import Any, NamedTuple

colorama.init()

@dataclass
class CarouselParameters:
 gravity: float = 9.81
 tilt_angle_deg: float = 30
 carousel_radius_m: float = 6
 gondola_mass_kg: float = 300
 carousel_rot_speed_rps: float = 0.2333
 gondola_width_m: float = 1.5
 gondola_height_m: float = 1.5

class DisplacementAnalysisResult(NamedTuple):
 global_minima: float | None
 global_maxima: float | None
 displacement_m: float | None
 criteria_met: bool

class SimulationResult(NamedTuple):
 spring_stiffness_n_per_m: float
 global_maxima_m: float | None
 global_minima_m: float | None
 displacement_m: float | None
 simulation_data: tuple

class CarouselSimulation:
 def __init__(
 self,
 params: CarouselParameters | None = None,
 simulation_time_s: float = 10,
 max_radial_displacement_m: float = 0.005,
) -> None:
 if params is None:
 params = CarouselParameters()

 self.params = params
 self.simulation_time_s = simulation_time_s
 self.max_radial_displacement_m = max_radial_displacement_m
 self.tilt_angle_rad = math.radians(params.tilt_angle_deg)
 self.spring_stiffness_n_per_m: float = 0

 self.initial_conditions: list[float] = [
 self.params.carousel_radius_m,
 0,
 0,
 2 * np.pi * self.params.carousel_rot_speed_rps,
```

```

]

def system_dynamics(self, _: float, state: list[float]) -> list[float]:
 position_m, velocity_ms, angle_rad, angular_velocity_rads = state

 dposition_dt = velocity_ms
 dvelocity_dt = (
 position_m * (angular_velocity_rads**2)
 + self.params.gravity * np.sin(self.tilt_angle_rad) * np.cos(angle_rad)
 +
 (self.spring_stiffness_n_per_m / self.params.gondola_mass_kg)
 * (self.params.carousel_radius_m - position_m)
)
)

dangle_dt = angular_velocity_rads
dangular_velocity_dt = -(
 2 * angular_velocity_rads * velocity_ms * position_m
 + self.params.gravity
 * np.sin(self.tilt_angle_rad)
 * np.sin(angle_rad)
 * position_m
) / (
 position_m**2
 + (5 / 3)
 * (self.params.gondola_width_m**2 + self.params.gondola_height_m**2)
 + 20 * self.params.carousel_radius_m**2
)

return [dposition_dt, dvelocity_dt, dangle_dt, dangular_velocity_dt]

def run_simulation(
 self,
) -> tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
 time_span_s = [0, self.simulation_time_s]
 solution = solve_ivp(
 self.system_dynamics,
 time_span_s,
 self.initial_conditions,
 method="RK45",
 rtol=1e-3,
 atol=1e-6,
)

 time_array_s = solution.t
 position_m, velocity_ms, angle_rad, angular_velocity_rads = solution.y
 return time_array_s, position_m, velocity_ms, angle_rad, angular_velocity_rads

def plot_results(self, simulation_data: tuple) -> None:
 time_array_s, position_m, velocity_ms, angle_rad, angular_velocity_rads = (
 simulation_data
)
 fig, axes = plt.subplots(4, 1, figsize=(10, 12))

 axes[0].plot(time_array_s, position_m, label=r"$x(t)$")
 axes[0].set_ylabel(r"x [m]")
 axes[0].legend(loc="upper right")
 axes[0].grid()
 axes[0].set_xlim(min(time_array_s), max(time_array_s))

 axes[1].plot(time_array_s, velocity_ms, label=r"$\dot{x}(t)$")
 axes[1].set_ylabel(r"\dot{x} [m/s]")
 axes[1].legend(loc="upper right")

```

```

 axes[1].grid()
 axes[1].set_xlim(min(time_array_s), max(time_array_s))

 angle_deg = np.degrees(angle_rad)
 axes[2].plot(time_array_s, angle_deg, label=r"$\alpha(t)$")
 axes[2].set_ylabel(r"α [deg]")
 axes[2].legend(loc="upper right")
 axes[2].grid()
 axes[2].set_xlim(min(time_array_s), max(time_array_s))

 angular_velocity_degs = np.degrees(angular_velocity_rads)
 axes[3].plot(time_array_s, angular_velocity_degs, label=r"$\dot{\alpha}(t)$")
 axes[3].set_ylabel(r"$\dot{\alpha}$ [deg/s]")
 axes[3].legend(loc="upper right")
 axes[3].grid()
 axes[3].set_xlim(min(time_array_s), max(time_array_s))

 for i in range(1, int(max(angle_deg) / 360) + 1):
 idx = np.where(np.isclose(angle_deg, 360 * i, atol=1))[0]
 if len(idx) > 0:
 axes[2].plot(time_array_s[idx[0]], angle_deg[idx[0]], "ro")
 for axis in axes:
 axis.axvline(
 x=time_array_s[idx[0]], color="r", linestyle="--", linewidth=1
)

 y_ticks = np.arange(0, max(angle_deg) + 360, 360)
 axes[2].set_yticks(y_ticks)
 axes[2].set_yticklabels([f"{int(y)}" for y in y_ticks])

 plt.tight_layout(rect=(0, 0.03, 1, 0.95))
 fig.text(0.5, 0.01, "Time (t)", ha="center", fontsize=12)
 fig.suptitle(
 f"Spring Stiffness={self.spring_stiffness_n_per_m} N/m", fontsize=16
)
 plt.show()

@staticmethod
def get_local_extremes(data_series: np.ndarray) -> tuple[list[float], list[float]]:
 local_maxima: list[float] = []
 local_minima: list[float] = []

 for i in range(1, len(data_series) - 1):
 if (
 data_series[i] > data_series[i - 1]
 and data_series[i] > data_series[i + 1]
):
 local_maxima.append(data_series[i])
 if (
 data_series[i] < data_series[i - 1]
 and data_series[i] < data_series[i + 1]
):
 local_minima.append(data_series[i])

 return local_minima, local_maxima

@staticmethod
def get_global_extremes(
 local_minima: list[float], local_maxima: list[float]
) -> tuple[float | None, float | None, float | None]:
 try:
 global_minima = round(min(local_minima), 5)
 global_maxima = round(max(local_maxima), 5)

```

```

difference = round(global_maxima - global_minima, 5)
return global_minima, global_maxima, difference

except ValueError:
 return None, None, None

def analyze_displacement(
 self, position_data: np.ndarray
) -> DisplacementAnalysisResult:
 local_minima, local_maxima = self.get_local_extremes(position_data)
 extremes = self.get_global_extremes(local_minima, local_maxima)
 global_minima, global_maxima, displacement_m = extremes

 criteria_met = False
 if (
 displacement_m is not None
 and displacement_m <= self.max_radial_displacement_m
):
 criteria_met = True

 return DisplacementAnalysisResult(
 global_minima=global_minima,
 global_maxima=global_maxima,
 displacement_m=displacement_m,
 criteria_met=criteria_met,
)

def print_simulation_result(
 self, analysis_result: DisplacementAnalysisResult
) -> None:
 global_minima = analysis_result.global_minima
 global_maxima = analysis_result.global_maxima
 displacement_m = analysis_result.displacement_m
 criteria_met = analysis_result.criteria_met

 if displacement_m is not None:
 message = (
 f"[Spring Stiffness: {self.spring_stiffness_n_per_m:<8} N/m] "
 f"Global maxima: {global_maxima:<8} m | "
 f"Global minima: {global_minima:<8} m | "
 f"Displacement: {displacement_m:<8} m"
)
 if criteria_met:
 print(colored(message, "green"))
 else:
 print(colored(message, "red"))

def process_stiffness_result(
 self,
 simulation_data: tuple,
 analysis_result: DisplacementAnalysisResult,
 show_results: bool,
) -> SimulationResult | None:
 if not analysis_result.criteria_met:
 return None

 result = SimulationResult(
 spring_stiffness_n_per_m=self.spring_stiffness_n_per_m,
 global_maxima_m=analysis_result.global_maxima,
 global_minima_m=analysis_result.global_minima,
 displacement_m=analysis_result.displacement_m,
 simulation_data=simulation_data,
)

```

```

 if show_results:
 self.plot_results(simulation_data)

 return result

 def _run_single_simulation(
 self, stiffness_value_n_per_m: float, show_results: bool
) -> SimulationResult | None:
 self.spring_stiffness_n_per_m = stiffness_value_n_per_m

 try:
 simulation_data = self.run_simulation()
 _, position_data, _, _, _ = simulation_data

 analysis_result = self.analyze_displacement(position_data)

 self.print_simulation_result(analysis_result)

 return self.process_stiffness_result(
 simulation_data,
 analysis_result,
 show_results,
)
 except ValueError as error:
 error_message = (
 f"[Spring Stiffness: {self.spring_stiffness_n_per_m:<8} N/m] "
 f"Value Error: {str(error)}"
)
 print(colored(error_message, "red"))
 return None

 except RuntimeError as error:
 error_message = (
 f"[Spring Stiffness: {self.spring_stiffness_n_per_m:<8} N/m] "
 f"Runtime Error: {str(error)}"
)
 print(colored(error_message, "red"))
 return None

 def find_optimal_spring_stiffness(
 self, search_config: dict[str, Any]
) -> list[SimulationResult]:
 start_n_per_m = search_config.get("start_n_per_m", 0)
 step_size_n_per_m = search_config.get("step_size_n_per_m", 100_000)
 num_results = search_config.get("num_results", 1)
 show_results = search_config.get("show_results", True)

 consecutive_valid_results = 0
 results_list: list[SimulationResult] = []
 max_stiffness = 999_999_999

 for stiffness in np.arange(start_n_per_m, max_stiffness, step_size_n_per_m):
 result = self._run_single_simulation(stiffness, show_results)

 if result:
 results_list.append(result)
 consecutive_valid_results += 1
 if consecutive_valid_results >= num_results:
 break
 else:
 consecutive_valid_results = 0

```

```
 return results_list

def main():
 simulator = CarouselSimulation()

 results = simulator.find_optimal_spring_stiffness(
 {
 "start_n_per_m": 0,
 "step_size_n_per_m": 100_000,
 "num_results": 1,
 "show_results": True,
 }
)
 return results

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

## Anhang H: case\_study\_results.csv

```
case_study,model,strategy,reliability_in_percent,optimization_time_in_s,input_tokens,output_tokens,optimization_costs_in_usd,functional_correctness_in_percent,ruleViolations,pylint_score,loc,sloc,avg_func_len_with_docstrings,avg_func_len_without_docstrings,commented_lines,avg_cc,mi
case_study_1,before_optimization,None,None,None,None,None,13.0,3.91,32.0,24.0,32.0,32.0,12.0,6.00,54.70
case_study_1,claude-3-7-sonnet-
20250219,ALL_AT_ONCE,100,12.9,12810,857,0.051,100,1.0,9.41,134.4,44.6,18.6,7.7,0.0,2.26,75.07
case_study_1,gemini-2-0-pro-exp-02-
05,ALL_AT_ONCE,100,9.9,12288,1092,0.021,100,0.0,9.95,165.0,56.2,30.7,12.6,1.0,3.32,72.17
case_study_1,gpt-4o-2024-08-
06,ALL_AT_ONCE,100,18.4,10806,572,0.033,100,2.8,9.03,115.4,39.8,19.7,10.0,1.0,2.50,75.41
case_study_1,grok-2-
1212,ALL_AT_ONCE,100,11.9,10912,585,0.028,100,3.2,8.13,120.0,34.0,17.5,7.2,0.4,2.30,77.09
case_study_1,claude-3-7-sonnet-
20250219,ONE_BY_ONE,100,63.9,17052,4558,0.120,100,0.0,10.00,143.2,42.8,32.7,11.3,1.0,2.67,75.07
case_study_1,gemini-2-0-pro-exp-02-
05,ONE_BY_ONE,100,96.7,17497,7094,0.057,100,0.4,10.00,217.0,58.0,29.4,8.3,0.2,2.05,72.38
case_study_1,gpt-4o-2024-08-
06,ONE_BY_ONE,100,109.1,14162,3309,0.068,100,0.0,10.00,103.8,41.8,19.3,9.4,1.0,2.80,75.28
case_study_1,grok-2-
1212,ONE_BY_ONE,100,121.5,15791,6117,0.093,100,2.0,7.38,234.8,48.8,24.7,5.2,0.4,1.82,73.78
case_study_2,before_optimization,None,None,None,None,None,8.0,7.10,87.0,82.0,15.2,15.2,6.0,3.40,42.42
case_study_2,claude-3-7-sonnet-
20250219,ALL_AT_ONCE,100,18.4,13249,1458,0.062,100,2.0,9.76,199.0,92.0,17.6,9.0,2.0,2.33,69.98
case_study_2,gemini-2-0-pro-exp-02-
05,ALL_AT_ONCE,100,15.4,12704,1878,0.025,100,0.8,10.00,262.0,92.8,21.0,8.8,3.0,2.26,68.10
case_study_2,gpt-4o-2024-08-
06,ALL_AT_ONCE,100,26.8,11163,939,0.037,100,2.0,9.85,141.4,73.2,23.8,14.2,2.0,3.40,71.15
case_study_2,grok-2-
1212,ALL_AT_ONCE,100,24.0,11263,1262,0.036,100,3.0,8.74,207.8,77.0,26.3,11.8,1.6,2.93,69.94
case_study_2,claude-3-7-sonnet-
20250219,ONE_BY_ONE,100,140.7,23233,12161,0.252,100,1.0,9.65,379.0,124.4,22.0,8.1,3.0,1.86,63.91
case_study_2,gemini-2-0-pro-exp-02-
05,ONE_BY_ONE,100,122.5,22595,12460,0.091,100,0.0,9.81,431.8,99.8,32.2,7.8,0.2,2.05,67.61
case_study_2,gpt-4o-2024-08-
06,ONE_BY_ONE,100,183.5,17625,8334,0.127,100,0.0,10.00,235.4,91.8,21.0,9.3,3.0,2.43,68.03
case_study_2,grok-2-
1212,ONE_BY_ONE,100,182.1,19082,10531,0.143,40,2.0,8.17,368.0,98.8,27.8,7.8,1.2,2.11,65.73
```

```
case_study_3,before_optimization,None,None,None,None,None,None,3.0,8.94,331.0,275.0,22
.7,22.7,1.0,2.56,24.79
case_study_3,claude-3-7-sonnet-
20250219,ALL_AT_ONCE,100,67.4,16138,5596,0.132,100,1.0,10.00,616.6,277.0,36.3,22.8,3.0,2.56
,50.72
case_study_3,gemini-2-0-pro-exp-02-
05,ALL_AT_ONCE,100,43.7,15474,5886,0.049,80,3.2,9.72,629.0,303.2,36.8,23.8,1.8,2.51,50.54
case_study_3,gpt-4o-2024-08-
06,ALL_AT_ONCE,100,105.8,13261,4078,0.074,100,1.2,9.80,539.4,277.8,32.6,22.9,3.2,2.66,48.69
case_study_3,grok-2-
1212,ALL_AT_ONCE,100,87.2,13411,4260,0.089,40,3.6,8.87,567.6,279.2,35.0,22.3,1.0,2.53,51.86
case_study_3,claude-3-7-sonnet-
20250219,ONE_BY_ONE,100,462.6,48432,38710,0.726,100,0.0,9.72,791.6,324.4,27.1,14.1,2.6,1.94
,49.27
case_study_3,gemini-2-0-pro-exp-02-
05,ONE_BY_ONE,100,269.8,47517,38554,0.252,100,3.8,9.78,823.8,376.8,35.7,20.7,1.4,2.14,47.75
case_study_3,gpt-4o-2024-08-
06,ONE_BY_ONE,100,523.6,34931,24953,0.337,100,0.0,9.93,562.6,286.0,35.1,22.8,3.0,2.56,52.45
case_study_3,grok-2-1212,ONE_BY_ONE,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

## Anhang I: GitHub-Repository

<https://github.com/phgas/optipy>