WHZ Westsächsische Hochschule Zwickau
Hochschule für Mobilität

Westsächsische Hochschule Zwickau

Fakultät für Physikalische Technik und Informatik

Wissenschaftliche Arbeit

Thema:

Nutzung von Sprachmodellen zur Verbesserung von Bezeichnern in Java-Programmen

Vorgelegt von: Kyz Saikal Tahirova
Studiengang: Informatik
Betreuer: Prof. Dr. Laue
Abgabetermin: 15.12.2025

Inhaltsverzeichnis

Kapitel 1

1 Einleitung

Sprachmodelle werden in der Softwareentwicklung immer häufiger eingesetzt, um Entwickler bei Analyse- und Review-Aufgaben zu unterstützen [10, 11]. Ein wichtiger Anwendungsbereich ist die Bewertung von Bezeichnern im Quellcode. Klare und verständliche Namen vereinfachen die Wartung, während verschwommene Bezeichner zu Fehlern und erhöhtem Aufwand führen [4].

Diese Arbeit untersucht die Zuverlässigkeit moderner Sprachmodelle bei der Einschätzung von Java-Bezeichnern. Bewertet werden Verständlichkeit, semantische Angemessenheit, Einhaltung etablierter Namenskonventionen [2] sowie die Erkennung linguistischer Anti-Muster [1]. Unter linguistischen Anti-Mustern versteht man systematische sprachliche Probleme in Bezeichnern, die Klarheit und Lesbarkeit beeinträchtigen.

Ziel der Untersuchung ist es zu prüfen, ob Sprachmodelle hilfreiche und konsistente Hinweise zur Verbesserung von Bezeichnern liefern können. Hierfür werden mehrere Hinweis-Varianten entwickelt und schrittweise verfeinert. Die Analyse erfolgt anhand von Java-Beispielen mit sowohl problematischen als auch korrekten Bezeichnern, um die Erkennung tatsächlicher Verstöße und das Auftreten unbegründeter Warnungen („False Positives“) zu bewerten. Frühere Arbeiten zeigen bereits, dass LLM-generierter Code zu semantischen Fehlbenennungen und strukturellen Qualitätsproblemen neigt [5, 8, 3].

Die zentrale Forschungsfrage lautet:

> Wie zuverlässig können Sprachmodelle Java-Bezeichner hinsichtlich Verständlichkeit, Bedeutung, Konventionen und linguistischer Anti-Muster bewerten?

Die Arbeit ist in mehrere Kapitel gegliedert:

1. **Kapitel 1 (Einleitung).** Vorstellung des Themas, Zielsetzung, Relevanz von Sprachmodellen in der Softwareentwicklung sowie die Forschungsfrage.
2. **Kapitel 2 (Theoretischer Hintergrund).** Überblick über Java-Bezeichner, Namenskonventionen und linguistische Anti-Muster.
3. **Kapitel 3 (Methodik und Durchführung).** Aufbau der Untersuchung, Entwicklung der Hinweis-Varianten sowie Beschreibung der verwendeten Java-Beispiele. *Eine automatische Pipeline wird nur theoretisch erwähnt.*
4. **Kapitel 4 (Evaluation).** Auswertung der Ergebnisse und systematische Bewertung der Antworten der Sprachmodelle.
5. **Kapitel 5 (Diskussion und Schluss).** Grenzen der Methode, Verbesserungsideen, Ausblick auf weitere Forschung.

Hinweis: In dieser ersten Version der Arbeit wird der automatische Pipeline-Ansatz nur theoretisch erläutert, da die Umsetzung zum jetzigen Zeitpunkt noch nicht erfolgt ist. In einer späteren Version könnte diese Funktion ergänzt werden, um den gesamten Prozess der Bezeichnerverbesserung automatisch abzubilden.

Kapitel 2

2 Theoretischer Hintergrund

Dieser Abschnitt bietet einen Überblick über Java-Bezeichner, etablierte Namenskonventionen sowie linguistische Anti-Muster, die als Grundlage für die spätere Analyse dienen. Die Relevanz dieser Konzepte ergibt sich aus empirischen Studien, die zeigen, dass Bezeichner einen erheblichen Einfluss auf die Verständlichkeit und Qualität von Quellcode haben [4].

2.1 Java-Bezeichner

Java-Bezeichner sind die Namen von Variablen, Methoden, Klassen, Schnittstellen, Konstanten und Paketen. Sie sollen die Funktion und den Zweck des Codes verständlich machen. Die Studie von Butler et al. [4] zeigt, dass unklare, inkonsistente oder mehrdeutige Namen mit geringer Wartbarkeit, erhöhter Komplexität und häufigeren Warnungen in statischen Analysewerkzeugen verbunden sind. Im Gegensatz dazu können klare Namen ein einfacher Indikator für eine hohe Qualität des Quellcodes sein.

2.2 Namenskonventionen in Java

Die offiziellen Java-Konventionen [2] empfehlen konsistente Schreibweisen und semantisch präzise Nomenphrasen. Dazu gehören:

- **Klassen und Schnittstellen.** Großschreibung jedes Wortanfangs (CamelCase), z. B. `CustomerAccount`, `PaymentProcessor`. Allgemein sollten aussagekräftige Nomenphrasen verwendet werden, etwa `EnrolledStudents` oder `NumberOfValidCreditCards`.
- **Methoden.** Verben oder Verbphrasen wie `calculateTotal()` oder `validateInput()`.
- **Attribute/Variablen.** camelCase und präzise Bedeutung, z. B. `userName`, `itemCount`.

Ein Name soll beschreiben, was ein Element speichert oder ausführt. Unklare Begriffe wie `tmp`, `data` oder `handle` lassen zu viel Interpretationsspielraum und führen schnell zu Missverständnissen.

2.3 Linguistische Anti-Muster

Linguistische Anti-Muster (Linguistic Anti-Patterns) wurden erstmals systematisch von Arnaoudova et al. beschrieben [1]. Sie bezeichnen wiederkehrende sprachliche Probleme in Bezeichnern, Kommentaren oder Signaturen, bei denen Name und tatsächliches Verhalten nicht übereinstimmen. Dies führt zu kognitiven Brüchen und erschwert das

Programmverständnis. Die linguistischen Anti-Muster lassen sich in zwei Hauptgruppen einteilen.

Methodenbezogene Anti-Muster:

1. *Kategorie A - tut mehr als der Name sagt.* Die Methode führt zusätzliche Aktionen aus. Beispiel: `getUser()` validiert oder speichert Daten.
2. *Kategorie B - sagt mehr als sie tut.* Der Name verspricht Funktionalität, die nicht implementiert ist. Beispiel: `validate()` besitzt den Rückgabetyp `void`.
3. *Kategorie C - tut das Gegenteil.* Name und Verhalten widersprechen sich. Beispiel: `disable()` erzeugt einen `EnableState`.

Attributbezogene Anti-Muster:

4. *Kategorie D - enthält mehr als der Name sagt.* Beispiel: ein als Prädikat benannter Bezeichner, dessen Typ jedoch kein `boolean` ist.
5. *Kategorie E - Name sagt mehr als enthalten ist.* Beispiel: `users` als Name für ein einzelnes Objekt.
6. *Kategorie F - Name und Inhalt widersprechen sich.* Beispiel: Attributname und Typ bilden Antonyme.

2.4 Große Sprachmodelle (LLMs)

Sprachmodelle werden zunehmend zur Bewertung von Quellcode verwendet [3]. Studien zeigen jedoch, dass LLMs zwar Muster erkennen können, aber gleichzeitig zu einer oberflächlichen Analyse neigen, insbesondere wenn nur Identifikatoren geändert oder semantisch irrelevante Änderungen vorgenommen werden [6]. Darüber hinaus zeigen mehrere Studien eine hohe Rate an Fehlalarmen („False Positives") und ungenauen Bewertungen [5, 9].

In dieser Arbeit wird untersucht, wie zuverlässig Sprachmodelle Java-Bezeichner bewerten können und in welchen Fällen weiterhin menschliches Eingreifen erforderlich ist.

Kapitel 3

3 Methodik und Durchführung

3.1 Auswahl der Sprachmodelle

Für die Untersuchung wurden zwei verschiedene Modelle über HAWKI[1] und ein Modell über Qwen KI abgefragt:

- OpenAI GPT-4o,
- Meta LLaMA 3.1 70B Instruct,
- Alibaba Qwen3-Coder.

3.2 Erstellung der Test-Beispiele

3.2.1 Kategorien nach Arnaoudova

Basierend auf Arnaoudova et al. [1] wurden zwei Arten von Beispielen erstellt:

- 13 Beispiele mit linguistischen Anti-Mustern (alle Kategorien A–F),
- 5 korrekt benannte Beispiele ohne Anti-Muster, um unbegründete Fehlalarme zu erkennen.

Jede Kategorie wurde kurz erklärt und durch ein prägnantes Beispiel illustriert.

3.2.2 Beispiel-Code

Kategorie A1 – „tut mehr als der Name sagt“. Methode mit `get` macht mehr als zurückgeben:

```
1  public class UserManager {
2      private User currentUser;
3
4      public User getUser() {
5          logAccess(); //Nebenwirkung
6          refreshSession(); //weitere Nebenwirkung
7          return currentUser;
8      }
9
10     private void logAccess() {}
11
12     private void refreshSession() {}
```

[1] HAWKI dient als Schnittstelle zu OpenAI GPT-4o, Meta LLaMA 3.1 70B Instruct

```
}
```

Kategorie B3. Name deutet Rückgabe an, aber `void`:

```
public class ConfigReader {
    public void getVersion() {
        System.out.println("v1.0");
    }
}
```

Kategorie C1. Widerspruch zwischen Name und Rückgabewert:

```
public class Config {
   public ControlState disable() {
        return new ControlState(true); // true = enabled
    }

    private static class ControlState {
        public boolean enabled;

        public ControlState(boolean enabled) {
            this.enabled = enabled;
        }
    }
}
```

Kategorie D2. Prädikatname, Typ nicht Boolean:

```
public class Product {
    public String isAvailable() {
        return "yes";
    }
}
```

Kategorie E1. Pluralname bei Typ im Singular:

```
public class CustomerService {
    private Customer customers;
}
```

Kategorie F1. Name und Typ im Gegensatz:

```
public class FeatureFlag {
    private boolean isEnabled = false;
}
```

3.2.3 Korrekte Beispiele für False-Positive-Tests

```java
public class CustomerService {

      private final CustomerRepository customerRepository;

      public CustomerService(CustomerRepository
         customerRepository) {
            this.customerRepository = customerRepository;
      }

      public Customer findCustomerById(Long customerId) {
            if (customerId == null || customerId <= 0) {
                  throw new IllegalArgumentException("Customer ID
                      must be positive");
            }
            return customerRepository.findById(customerId);
      }
}

public interface CustomerRepository {
        Customer findById(Long id);
        void save(Customer customer);
        void deleteById(Long id);
    }
```

Diese Beispiele enthalten keine Anti-Muster. Ein korrektes Modell darf hier *keine* Warnung ausgeben [9].

3.3 Hinweis-Strategien und Iteration

Die Hinweise wurden iterativ nach Versionen überarbeitet: Version 1 (V1) → Version 2 (V2) → Version 3 (V3) → Version 4 (V4). Das Ziel jeder Iteration war die Qualitätssteigerung der Hinweise (automatischer Überblick) und die Reduzierung der Anzahl von Fehlalarmen. V4 enthält auch eine kurze Definition linguistischer Anti-Muster, damit Modelle diese Kategorien zielorientiert erkennen können. Die Beschreibung der Kategorien A–F fällt im Prompt bewusst knapp aus und fasst D–F nur zusammen als „Inkonsistenzen bei Attributnamen“. Eine sinnvolle Verbesserungsmöglichkeit für zukünftige Arbeiten wäre, jede Kategorie A–F im Hinweis mit einer eigenen klaren Definition und einem kurzen Beispiel zu erklären, damit LLM alle Arten von Anti-Mustern besser unterscheiden kann.

V1 (Zero-Shot)

```
Analysieren Sie die Bezeichner im folgenden Java-Code
und identifizieren Sie mögliche Benennungs- oder Struktur-Anti-Muster.
[CODE]
```

V2 (Few-Shot)

```
Beispiele für gute Bezeichner:
- Klassen: CustomerAccount, PaymentProcessor
- Methoden: calculateTotal(), validateInput()
- Attribute: userName, itemCount
Analysieren Sie nun folgenden Code:
[CODE]
```

V3 (Kontextreicher Hinweis)

```
Analysieren Sie den Code unter Berücksichtigung von:
1) Java-Konventionen (Oracle)
2) Konsistenz zwischen Name, Typ und Verhalten
3) Verständlichkeit und Bedeutung
Geben Sie Hinweise für Entwickler.
[CODE]
```

V4 (Erklärung der linguistischen Anti-Muster)

Da unklar ist, ob Modelle linguistische Anti-Muster kennen [1], wird in dieser Version eine kurze Erklärung ergänzt.

```
Linguistische Anti-Muster:
A: Methode tut mehr als Name sagt
B: Name verspricht mehr als implementiert
C: Name und Verhalten sind gegensätzlich
D-F: Inkonsistenzen bei Attributnamen
Analysieren Sie nun den Code:
[CODE]
```

3.4 Iteratives Vorgehen

Der Hinweis wurde iterativ verbessert:

1. erste Tests mit Version 1,

2. Analyse der Fehlbewertungen,
3. Erweiterung um Beispiele (Version 2),
4. Hinzufügen von Kontextregeln (Version 3),
5. Ergänzung der Definitionen von Anti-Mustern (Version 4).

Dieses Vorgehen folgt aktuellen Forschungsergebnissen, die die Bedeutung von Hinweis-Gestaltung und Iteration bei der Codeanalyse hervorheben [8, 3].

3.5 Durchführung der Tests

Für jedes der 13 fehlerhaften Beispiele und für die 5 korrekten Beispiele wurde jede Hinweis-Version an alle drei Modelle über HAWKI gesendet. Jede Modellantwort wurde protokolliert und gespeichert.

3.6 Bewertungskriterien

Jedes Kriterium wird mit 1-5 Punkten bewertet.

1. **Anti-Muster-Erkennung.** Wurde das Problem erkannt (Ja/Nein)?
2. **Korrekturqualität.** Ist der vorgeschlagene neue Name besser? Löst er das Anti-Muster?
3. **Konventions-Konformität.** Folgt der neue Code Java-Standards?
4. **Semantische Klarheit.** Sind die neuen Namen verständlich, und vermeiden die neu vorgeschlagenen Namen linguistische Anti-Muster?
5. **Konsistenz.** Passt alles zusammen?

Zusätzlich wurden Fehlalarme („False Positives“) gesondert untersucht, da aktuelle Forschung zeigt, dass sie bei LLM-basierten Analysen häufig auftreten [9, 6].

Kapitel 4

4 Evaluation

Die Untersuchung nutzt insgesamt 18 Java-Beispiele: 13 Beispiele mit linguistischen Anti-Mustern der Kategorien A–F und 5 korrekte Beispiele ohne Anti-Muster. Jedes Beispiel wurde mit vier Hinweisversionen (V1–V4) an drei Modelle getestet: OpenAI GPT-4o, Meta LLaMA 3.1 70B Instruct und Alibaba Qwen3-Coder. Damit ergeben sich 216 Modellantworten (18 Beispiele × 4 Versionen × 3 Modelle).

4.1 Ergebnisse nach Modellen

Tabelle 4.1 zeigt die wichtigsten Unterschiede zwischen den Modellen.

Tabelle 4.1: Vergleich der Modelleigenschaften

Aspekt	GPT-4o	LLaMA 3.1 70B	Qwen3-Coder
Anti-Muster-Nennung (V1–V3)	Selten	Selten	Selten
Anti-Muster-Nennung (V4)	Häufig, korrekt	Häufig, überinterpretiert	Häufig, korrekt
Fehlalarme	Keine	Mehrere (alle Versionen)	Keine
Durchschnitt (V1–V3)	4,17	4,29	3,75
Durchschnitt (V4)	5,00	4,88	5,00

4.1.1 OpenAI GPT-4o

GPT-4o zeigt in allen Beispielen eine stabile Leistung. In den Hinweisversionen V1 bis V3 beschreibt das Modell die Probleme präzise, verwendet die Anti-Muster-Begriffe jedoch nur selten, etwa wenn es in Beispiel A1 die Nebenwirkungen von `getUser()` kritisiert, den Fall aber nicht Kategorie A zuordnet (Anti-Muster gefunden = nein, Gesamtpunktzahl 4,00).

Mit Hinweisversion V4 ändert sich dieses Verhalten deutlich: GPT-4o nennt die Kategorien A–F explizit und erreicht bei allen 13 fehlerhaften Beispielen eine Bewertung von 5,00; in Beispiel A1 wird Anti-Muster A und in Beispiel C1 werden die Kategorien B und C klar identifiziert.

Bei den korrekten Beispielen bestätigt GPT-4o konsequent die Qualität der Bezeichner, erzeugt keine Fehlalarme und vergibt durchgängig Bewertungen von 5,00 Punkten.

4.1.2 Meta LLaMA 3.1 70B Instruct

Meta LLaMA 3.1 70B Instruct zeigt gemischte Ergebnisse. Bei fehlerhaften Beispielen liefert das Modell in den Versionen V1 bis V3 gute Analysen mit Gesamtnoten zwischen 3,75 und 4,75, nutzt die Anti-Muster-Begriffe jedoch erst ab Version V4 konsequent. In dieser Version erreicht LLaMA bei mehreren Beispielen (u. a. D1, D2, E1 und E2) jeweils 5,00 Punkte und ordnet die Probleme explizit den Kategorien B, C oder D zu.

Problematisch ist das Verhalten bei korrekten Beispielen. Im Beispiel D1 behauptet LLaMA in allen Versionen, der Code müsse verbessert werden, und schlägt etwa eine Umbenennung von `isPremium` zu `hasPremiumMembership` vor, obwohl hier kein Anti-Muster vorliegt (Fehlalarme, Note zwischen 4,25 und 4,50).

4.1.3 Alibaba Qwen3-Coder

Alibaba Qwen3-Coder verhält sich in vielen Punkten ähnlich wie GPT-4o. In den Versionen V1 bis V3 beschreibt das Modell Probleme korrekt, nutzt die Anti-Muster-Kategorien jedoch kaum; die Bewertungen liegen dabei konstant bei etwa 3,75. Typische Kommentare lauten, dass ein Name wie `checkFormat()` ungenau ist oder dass er Unstimmigkeiten beschreibt, die keiner bestimmten Anti-Muster-Kategorie zuzuordnen.

Mit Version V4 steigt die Leistung von Qwen3-Coder: in allen betrachteten Beispielen (unter anderem B1, B2, B3, F1 und F2) erreicht das Modell eine Bewertung von 5,00 und ordnet die Probleme den passenden Kategorien zu; in F1 werden Anti-Muster im Bereich B und D–F explizit identifiziert; in F2 Anti-Muster D.

Bei den korrekten Beispielen verhält sich Qwen3-Coder zuverlässig, erzeugt keine Fehlalarme und bestätigt eine korrekte Bezeichnerpraxis; die Bewertungen liegen auch durchgängig bei 5,00 Punkten.

4.2 Vergleich der Hinweisversionen

Version 1 (Zero-Shot). Ohne Beispiele und ohne Definition der Anti-Muster liefern alle Modelle zwar sinnvolle Kommentare und erkennen semantische Probleme, nennen die Kategorien A–F jedoch fast nie explizit. Die durchschnittlichen Bewertungen liegen bei etwa 3,75 bis 4,00 für GPT-4o, 3,75 bis 4,75 für LLaMA 3.1 und 3,00 bis 3,75 für Qwen3-Coder; Anti-Muster werden nicht erkannt.

Version 2 (Few-Shot). Die Beispiele für gute Bezeichner erhöhen die Korrekturqualität leicht, sodass die Bewertungen teils von 3,75 auf 4,00 steigen. Die Modelle verwenden die Anti-Muster-Begriffe jedoch weiterhin selten und benennen Probleme ohne klare Zuordnung zu Kategorien wie A oder C.

Version 3 (kontextreich). Der Kontext zu Java-Konventionen und Konsistenz führt zu strukturierteren Analysen mit hohen Werten bei Klarheit und Konsistenz (oft 5 Punkte). Die Anti-Muster-Kategorien werden jedoch nicht systematisch verwendet, sodass Fälle meist nur implizit beschrieben werden.

Version 4 (mit Anti-Muster-Definitionen). Version V4 mit expliziter Anti-Muster-Definition verbessert die Leistung aller Modelle deutlich: GPT-4o und Qwen3-Coder erreichen bei allen 13 fehlerhaften Beispielen 5,00 Punkte mit korrekter Kategorienzuordnung; LLaMA 3.1 erzielt ebenfalls 5,00 bei fehlerhaften Fällen, zeigt aber weiterhin Fehlalarme bei korrektem Code.

Die Hinweis-Gestaltung beeinflusst die Modelle stark - ohne explizite Definition erkennen sie Probleme inhaltlich, nutzen aber nicht die Anti-Muster-Begriffe; Version V4 schließt diese Lücke weitgehend.

4.3 Analyse der Fehlalarme (False Positives)

Die Untersuchung analysierte Fehlalarme bei korrektem Code. GPT-4o und Qwen3-Coder erzeugen keine Fehlalarme und bestätigen konsequent die Qualität der Bezeichner (Bewertung 5,00).

LLaMA 3.1 70B meldet hingegen systematisch Probleme, etwa bei D1 in allen Versionen: V1–V3 kritisieren ungerechtfertigt, V4 ordnet fälschlich Anti-Muster B/D zu (Bewertung 4,25–4,50). V4 verstärkt bei LLaMA die Überinterpretation, verbessert aber bei GPT-4o und Qwen3-Coder die Erkennung ohne Spezifitätsverlust.

4.4 Zusammenfassung der Evaluationsergebnisse

Die Evaluation führt zu fünf zentralen Erkenntnissen:

- **Version V4 ist entscheidend**. Erst die explizite Definition der Anti-Muster führt dazu, dass alle Modelle die Kategorien A–F systematisch nutzen; die Bewertungen steigen von durchschnittlich 3,75–4,75 (V1–V3) auf 5,00 (V4) bei fehlerhaften Beispielen.
- **GPT-4o und Qwen3-Coder zeigen vergleichbare Leistung**. Beide erreichen mit V4 eine Erkennungsrate von 100 % bei fehlerhaften Beispielen ohne Fehlalarme bei korrekten Fällen, und die Vorschläge sind in allen Kriterien hervorragend.
- **LLaMA neigt zu Fehlalarmen**. Trotz guter Ergebnisse bei fehlerhaften Beispielen meldet LLaMA systematisch Probleme bei korrektem Code, und dieses Verhalten bleibt in allen Versionen bestehen.

- **Inhaltsanalyse ohne Anti-Muster-Begriffe**. Bereits V1–V3 erkennen alle Modelle inhaltliche Probleme wie Nebenwirkungen oder Widersprüche, auch ohne Anti-Muster-Begriffe.
- **Hohe Konventionskonformität**. Über alle Modelle und Versionen liegen die Werte für Konventionskonformität und Klarheit bei 4–5 Punkten; die Vorschläge folgen durchgängig Java-Standards.

Sprachmodelle erkennen Anti-Muster gut, brauchen aber klare und präzise Anweisungen. Zusätzlich wurde überprüft, ob die von den Modellen vorgeschlagenen neuen Namen selbst keine linguistischen Anti-Muster enthalten. In den meisten Fällen lösen die Vorschläge von GPT-4o und Qwen3-Coder das Ausgangsproblem und führen nicht zu neuen Widersprüchen zwischen Name, Typ und Verhalten, während LLaMA 3.1 in einigen Fällen neue semantische Ungenauigkeiten oder inkonsistente Attributnamen generiert. Die Fehlalarme bei LLaMA zeigen deutlich: gute Auffindung bedeutet nicht automatisch korrekte Ergebnisse.

Kapitel 5

5 Diskussion und Schluss

Diese Arbeit untersucht die Zuverlässigkeit von Sprachmodellen bei der Bewertung von Java-Bezeichnern. Vier Hinweisversionen (V1–V4) wurden an 18 Beispielen mit drei Modellen (GPT-4o, LLaMA 3.1 70B, Qwen3-Coder) getestet.

Zentrale Ergebnisse:

- Explizite Anti-Muster-Definition in V4 verbessert die Erkennungsrate dramatisch.
- GPT-4o und Qwen3-Coder erreichen 100 % Erkennung ohne Fehlalarme.
- LLaMA erzeugt systematisch Fehlalarme bei korrektem Code.
- Vorschläge sind konventionskonform und direkt einsetzbar.

5.1 Kritische Reflexion

Die Untersuchung ist begrenzt:

- Nur 18 Beispiele ohne komplexe Projektkontexte.
- Manuelle Bewertung statt automatischer Pipeline.
- Nur drei Modelle, keine Langzeit-Konsistenz geprüft.

Außerdem erklärt der verwendete Hinweis in Version V4 die Kategorien D–F nur sehr grob. In zukünftigen Studien sollten detailliertere Beschreibungen der Hinweise für die einzelnen Kategorien A–F untersucht werden, um die Erkennung verschiedener Anti-Muster weiter zu verbessern.

Trotzdem zeigen GPT-4o und Qwen3-Coder mit V4 hohe praktische Nutzbarkeit. Die Qualität hängt stark von der Hinweis-Formulierung ab. Sprachmodelle sind wertvolle Assistenten, ersetzen aber keine menschliche Überprüfung.

Literatur

[1] Arnaoudova, V., Di Penta, M., & Antoniol, G. (2016). Linguistic Antipatterns: What They Are and How Developers Perceive Them. *Empirical Software Engineering*, 21(1), 104–158. DOI: https://doi.org/10.1007/s10664-014-9350-8.

[2] Oracle Corporation. (o.J.). *Naming Conventions*. Verfügbar unter: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html (abgerufen am 14.10.2025).

[3] Lyu, M. R., Ray, B., Roychoudhury, A., Tan, S. H., & Thongtanunam, P. (2025). Automatic Programming: Large Language Models and Beyond. *ACM Transactions on Software Engineering and Methodology*, 34(5), Article 140, 33 Seiten. DOI: https://doi.org/10.1145/3708519.

[4] Butler, S., Wermelinger, M., Yu, Y., & Sharp, H. (2010). Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In: *14th European Conference on Software Maintenance and Reengineering (CSMR)*, 15–18 March 2010, Madrid, Spain, pp. 156–165. IEEE. DOI: https://doi.org/10.1109/CSMR.2010.27.

[5] Ghosh Paul, D., Zhu, H., & Bayley, I. (2025). Investigating the Smells of LLM Generated Code. *arXiv preprint*. DOI: https://doi.org/10.48550/arXiv.2510.03029.

[6] Santa Molison, A., Moraes, M., Melo, G., Santos, F., & Assunção, W. K. G. (2025). Is LLM-Generated Code More Maintainable & Reliable than Human-Written Code? *arXiv preprint*. DOI: https://doi.org/10.48550/arXiv.2508.00700.

[7] Gnieciak, D., & Szandala, T. (2025). Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection. *arXiv preprint*. DOI: https://doi.org/10.48550/arXiv.2508.04448.

[8] Akram, W., Jiang, Y., Zhang, Y., Khan, H. A., & Liu, H. (2025). LLM-Based Method Name Suggestion with Automatically Generated Context-Rich Prompts. *Proceedings of the ACM on Software Engineering*, 2(FSE), Article FSE036, 22 Seiten. DOI: https://doi.org/10.1145/3715753.

[9] Andrade, R., Torres, J., & Ortiz-Garcés, I. (2025). Enhancing Security in Software Design Patterns and Antipatterns: A Framework for LLM-Based Detection. *Electronics*, 14(3), 586. DOI: https://doi.org/10.3390/electronics14030586.

[10] He, J., Shi, J., Zhuo, T. Y., Lo, D., et al. (2025). From Code to Courtroom: LLMs as the New Software Judges. *arXiv preprint*. DOI: https://doi.org/10.48550/arXiv.2503.02246.

[11] Velasco, A., Rodriguez-Cardenas, D., Khati, D., Palacio, D. N., Alif, L. R., & Poshyvanyk, D. (2026). A Causal Perspective on Measuring, Explaining and Mitigating Smells in LLM-Generated Code. In: *48th IEEE/ACM International Conference on Software Engineering (ICSE '26)*, 12–18 April 2026, Rio de Janeiro, Brazil, ACM, 12 Seiten. DOI: https://doi.org/10.1145/3744916.3773164.