Informatik-Unterricht im Zeitalter der KI

Brauchen wir noch Programmierunterricht, wenn KI programmieren kann?

Autor: Thomas Jörg

1 Die Kernfrage

Im Februar 2025 prägte Andrej Karpathy – Mitgründer von OpenAI, ehemaliger KI-Chef bei Tesla und einer der einflussreichsten Denker im Bereich der Künstlichen Intelligenz – einen Begriff, der die Tech-Welt elektrisierte: **Vibe Coding**. Seine These: Man gibt sich den Vibes hin, vergisst, dass der Code überhaupt existiert, und lässt die KI programmieren. Die natürliche Sprache – Englisch, Deutsch, Spanisch – sei die wichtigste Programmiersprache unserer Zeit.

Parallel argumentierte Matt Welsh, ehemaliger Informatik-Professor in Harvard, noch radikaler: Programmieren sei dem Untergang geweiht. Software werde künftig nicht mehr programmiert, sondern trainiert. LLMs seien virtuelle Maschinen, die in natürlicher Sprache programmiert werden.

Für Informatik-Lehrkräfte stellt sich damit eine existenzielle Frage: Wenn KI Code schreiben kann – warum sollten Schülerinnen und Schüler noch lernen, es selbst zu tun? Ist der Syntax-Drill in Java, Python und SQL noch gerechtfertigt? Oder verschwenden wir wertvolle Unterrichtszeit auf Fähigkeiten, die morgen ein KI-Agent in Sekunden erledigt?

Die zentrale These lautet: **Programmieren bleibt essenziell – aber die Begründung verschiebt sich grundlegend.** Nicht mehr das Schreiben von Code, sondern das Verstehen, Bewerten und Orchestrieren steht im Mittelpunkt.

2 Abstraktionsschichten: Warum jede Stufe die Informatik vorangebracht hat

Die Geschichte der Informatik lässt sich als eine Abfolge von Abstraktionsschichten lesen, von denen jede einzelne Kognition freigesetzt hat:

Ära	Abstraktionsschicht	Was wurde „delegiert“?	Was wurde dadurch möglich?
1940er	Maschinencode	–	Direkte Hardware-Steuerung
1950er	Assembler	Binäre Opcodes	Lesbare Mnemonics, Wiederverwendung
1960er	Hochsprachen (Fortran, C)	Register, Speicheradressen	Algorithmen-Design, Portabilität
1980er	OOP (C++, Java)	Speicherverwaltung, Modularität	Systemmodellierung, Kapselung
2000er	Frameworks (Spring, Django)	Boilerplate, Infrastruktur	Geschäftslogik, Rapid Prototyping
2020er	**KI-Agenten (LLMs)**	**Syntax, Implementierung**	**Architektur, Konzeption, Bewertung**

Der entscheidende Punkt: **Niemand hat jemals gefordert, weiterhin Assembler zu unterrichten, „damit Schüler wirklich verstehen, was ein Computer tut“.** Stattdessen hat jede Abstraktionsschicht es ermöglicht, sich auf höherwertige Probleme zu konzentrieren. Die Einführung von Compilern hat das

Verständnis von Algorithmen nicht überflüssig gemacht – sie hat es überhaupt erst ermöglicht, sich auf Algorithmen zu konzentrieren, statt auf Register und Speicheradressen.

LLMs sind die nächste Abstraktionsschicht. Und genau wie bei Compilern gilt ein doppeltes Prinzip: Wer die darunterliegende Schicht gar nicht versteht, wird scheitern. Aber wer seine gesamte Unterrichtszeit auf der darunterliegenden Schicht verbringt, verpasst die Konzepte, die jetzt erreichbar werden.

> **Aus der Unterrichtspraxis**
> Ein konkretes Beispiel: In der gymnasialen Oberstufe Baden-Württemberg lernen Schülerinnen und Schüler Java, SQL und Python. Abitursaufgaben in Java erfordern enormen Zeitaufwand für Syntax-Details wie die Unterscheidung von == und .equals(). Diese Unterrichtszeit fehlt für vertiefte Graphentheorie, ML-Konzepte oder Architekturdenken – Kompetenzen, die in einer KI-gestützten Arbeitswelt deutlich relevanter sind.

3 Die Debatte: Zwei Positionen, zwei Welten

3.1 Position A: „Natürliche Sprache ist die wichtigste Programmiersprache“

> *„There’s a new kind of coding I call ‘vibe coding’, where you fully give in to the vibes, embrace exponentials, and forget that the code even exists.“*
>
> — Andrej Karpathy, X/Twitter, Februar 2025

Karpathys These, geäußert als „throwaway tweet“, traf einen Nerv. Innerhalb weniger Wochen wurde Vibe Coding zum Leitbegriff einer Bewegung. Die Kernargumente:

Natürliche Sprache als Interface: Durch LLMs können Computer unsere Muttersprache direkt verstehen. Man beschreibt sein Ziel in Deutsch oder Englisch, und die KI implementiert es. Komplexe Syntax, geschlossene Klammern und Typ-Deklarationen werden zur Aufgabe der Maschine.

Radikale Beschleunigung: Aufgaben, die früher ein Wochenende dauerten – Server einrichten, Modelle testen, eine Web-Oberfläche bauen – erledigen KI-Agenten in 30 Minuten. Sie recherchieren Lösungen, debuggen, testen und liefern einen fertigen Bericht.

Matt Welsh (ehem. Harvard) geht weiter: Software werde künftig nicht mehr programmiert, sondern LLMs würden Aufgaben direkt ausführen. Die Informatik als Disziplin stehe vor einem fundamentalen Umbruch.

Die Zahlen scheinen ihnen Recht zu geben: 25 % der Startups im Y-Combinator-Batch Winter 2025 hatten Codebases, die zu 95 % KI-generiert waren. Google berichtet, dass etwa ein Viertel ihres Codes bereits KI-unterstützt entsteht. 84 % der Entwickler nutzen oder planen den Einsatz von KI-Tools (Stack Overflow Developer Survey 2025).

3.2 Position B: „Ohne Fundamentals geht es nicht“

> *„KI versteht nicht, warum etwas so ist, wie es ist. Sie kennt keine Zielkonflikte zwischen Performance und Wartbarkeit. Sie hat kein Gespür dafür, wie eine Architektur zur Organisation passt.“*
>
> — Golo Roden, Heise Online, Januar 2026

Roden, Gründer und CTO eines Webentwicklungsunternehmens, formuliert das Gegenargument prägnant: Die Wertschöpfung in der Softwareentwicklung verschiebt sich – weg von der reinen Umsetzung, hin zur Konzeption. KI arbeitet auf der Ebene von Syntax und Mustern. Sie reproduziert wiederkehrende Strukturen. Für Standardaufgaben reicht das. Aber für die Entscheidungen, die später schwer zu ändern sind – Service-Schnitte, Abhängigkeiten, Modulgrenzen – braucht man Kontextwissen, Erfahrung und Urteilsvermögen.

Andrew Ng, einer der weltweit führenden KI-Forscher, argumentiert differenzierter: Gerade weil KI programmieren kann, sei es wichtiger denn je, die Grundlagen zu verstehen. Die zunehmende Vereinfachung des Programmierens sollte dazu führen, dass **mehr** Menschen programmieren lernen, nicht weniger. Wer die Grundlagen beherrscht, kann KI-Tools effektiver nutzen, Fehler erkennen und die Technologie an eigene Bedürfnisse anpassen.

Die **Gesellschaft für Informatik (GI)** und die **TeachAI-Initiative** (unter Mitwirkung der GI) veröffentlichten im Juli 2025 einen Leitfaden zur Zukunft der Informatikausbildung. Kernbotschaft: Programmieren bleibt essenziell, aber der Fokus muss sich verschieben – weg vom reinen Codieren, hin zum Verstehen, wie KI die Informatik verändert.

4 Karpathys Wende: Vom Vibe Coding zum Agentic Engineering

Der aufschlussreichste Aspekt dieser Debatte ist nicht die Auseinandersetzung zwischen Lagern, sondern die **Evolution desselben Denkers**. Karpathys eigene Entwicklung in nur zwölf Monaten liefert das stärkste Argument für die Notwendigkeit von Fundamentals:

Februar 2025: Die Euphorie

Karpathy prägt „Vibe Coding“ als einen Modus, in dem man der KI vollständig vertraut. Code entsteht durch Beschreiben, Ausprobieren, Kopieren. Syntax ist irrelevant. Karpathy beschreibt es als besonders geeignet für schnelle Wochenendprojekte.

Frühling/Sommer 2025: Die Ernüchterung

Karpathy versucht, sein Projekt „Nanochat“ mit Vibe Coding zu bauen – und scheitert. LLMs haben leicht veraltetes Wissen, machen subtile aber kritische Designfehler, und halluzinieren Lösungen. Er programmiert das Projekt schließlich von Hand.

Dezember 2025: Das Erdbeben

> *„I’ve never felt this much behind as a programmer. Clearly some powerful alien tool was handed around except it comes with no manual and everyone has to figure out how to hold it and operate it, while the resulting magnitude 9 earthquake is rocking the profession.“*
>
> — Andrej Karpathy, Dezember 2025

KI-Agenten überqueren im Dezember 2025 eine Leistungsschwelle. Was Karpathy als „Erdbebenstärke 9“ für den Berufsstand beschreibt, löst eine Produktivitätspanik in der Tech-Branche aus.

Februar 2026: Die Reifung

Exakt ein Jahr nach dem Vibe-Coding-Tweet erklärt Karpathy den Begriff für überholt. Der neue Leitbegriff: **Agentic Engineering**. Die Kernaussage: Man schreibt zu 99 % nicht mehr selbst Code, sondern orchestriert KI-Agenten und überwacht deren Arbeit. Entscheidend ist Karpathys Begründung für die Wortwahl: „**Engineering**, um zu betonen, dass es eine Kunst und Wissenschaft und Expertise dafür gibt. Es ist etwas, das man lernen und worin man besser werden kann, mit einer eigenen Tiefe."

Die Lektion für den Unterricht

Karpathys Wende in 12 Monaten – von „vergiss den Code" zu „es gibt eine Kunst und Wissenschaft" – zeigt: Selbst der größte Enthusiast für KI-gestützte Programmierung kommt zum Schluss, dass Expertise, Urteilsvermögen und systematisches Vorgehen unverzichtbar sind. Die Frage ist nicht ob Fundamentals nötig sind, sondern welche.

5 Die Evidenzlage 2025/2026: Der Vibe Coding Hangover

Die anfängliche Euphorie über KI-generierte Software wich im Laufe des Jahres 2025 einer ernüchterten Bestandsaufnahme. Die Daten sind eindeutig:

5.1 Sicherheit und Qualität

Quelle	Befund
Veracode (2025)	**45 %** des KI-generierten Codes enthält Sicherheitslücken
Uplevel Research (2025)	KI-unterstützter Code enthielt **41 %** mehr Bugs als manuell geschriebener
Cloud Security Alliance	**62 %** des KI-generierten Codes enthält Sicherheitsmängel
Stanford-Universität	Entwickler mit KI-Assistenten schrieben **unsichereren** Code, glaubten aber, er sei **sicherer**
Stack Overflow (2025)	Über **ein Drittel** der Besuche resultieren aus Problemen mit KI-generiertem Code

Das Vertrauens-Paradox (Stanford-Studie)

Dies ist vielleicht der beunruhigendste Befund: Je weniger jemand den Code versteht, den die KI produziert, desto mehr vertraut er ihm. Entwickler, die KI-Assistenten nutzten, produzierten signifikant unsichereren Code – bewerteten ihn aber als sicherer. Für den Unterricht bedeutet das: Ohne Grundverständnis entsteht gefährliches Scheinwissen.

5.2 Das Junior-Developer-Paradox

Stanford-Daten zeigen einen Rückgang von 16 % bei Stellenanzeigen für junge Software-Entwickler. Gleichzeitig planen laut einer LeadDev-Umfrage 54 % der Engineering-Leiter, weniger Junior-Entwickler einzustellen.

Die Ironie ist offensichtlich: **Genau die Erfahrung, die man zum Bewerten von KI-Output braucht, wird durch die Eliminierung von Einstiegspositionen vernichtet.** Die Ingenieure, die 2027 KI-generierte Altlasten aufräumen müssen, werden nicht existieren, weil sie 2025 nicht eingestellt wurden.

Für den Informatik-Unterricht ist das ein Warnsignal: **Wenn die Industrie weniger Gelegenheit bietet, durch Praxis zu lernen, muss die Schule mehr konzeptuelles Fundament legen – nicht weniger.**

5.3 Praxis-Schocks

Im März 2025 baute ein Startup-Gründer seine gesamte SaaS-Anwendung mit KI-Tools – ohne eine Zeile eigenen Codes. Zwei Tage nach dem Launch brach alles zusammen: API-Schlüssel waren im Frontend exponiert, die Datenbank war ungeschützt, Nutzer konnten die Bezahlschranke umgehen. Im Juli 2025 löschte ein KI-Coding-Agent die Produktionsdatenbank eines Nutzers – trotz expliziter Anweisung, dies nicht zu tun.

Diese Fälle sind keine Einzelfälle. Sie illustrieren ein strukturelles Problem: **KI-generierter Code kann funktionieren, ohne korrekt, sicher oder wartbar zu sein.** Wer den Unterschied nicht erkennen kann, ist gefährdet.

6 Synthese: Der Kompetenz-Kompass

Aus der Analyse der Debatte und der Evidenzlage ergibt sich ein klares Bild. Die Frage ist nicht „Programmieren ja oder nein?“, sondern: **Welches Programmieren?** Die folgende Übersicht sortiert die zentralen Kompetenzen des Informatik-Unterrichts neu.

6.1 Was bleibt – und wichtiger wird

Kompetenz	Warum wichtiger denn je	Bezug zum klassischen Curriculum
Algorithmisches Denken Dekomposition, Abstraktion, Mustererkennung, Generalisierung	Karpathys „Agentic Engineering“ erfordert genau diese Fähigkeiten: Ein Problem so zerlegen, dass KI-Agenten es lösen können	Algorithmen & Datenstrukturen, Theoretische Informatik
Formale Modellierung Zustandsmaschinen, Grammatiken, Graphen als Denkwerkzeuge	Formale Beschreibung von Problemen ist die Grundlage für präzise KI-Prompts und Context Engineering	Automatentheorie, Graphentheorie, Formale Sprachen
Systemdenken & Architektur Abhängigkeiten, Modulgrenzen, Entwurfsentscheidungen	Rodens Kernargument: KI kann Code produzieren, aber nicht entscheiden, wie ein System geschnitten werden soll	Software-Engineering, Datenbank-Design (ER, Normalisierung)
Kritische Bewertungskompetenz Code lesen, verstehen, bewerten, testen	Stanford-Studie: Ohne diese Kompetenz entsteht gefährliches Scheinwissen; die Fehlerquote bei KI-Code ist hoch	Debugging, Testen, Code Review
Datenmodellierung ER-Konzepte, Normalisierung, relationale Logik	KI generiert Datenbankschemas, die technisch funktionieren, aber semantisch falsch sein können (z. B. fehlende Normalisierung, falsche Beziehungstypen)	Relationale Datenbanken, SQL (konzeptuell)

6.2 Was reduziert werden kann

Bisher im Unterricht	Warum reduzierbar	Stattdessen
Syntax-Drill in mehreren Sprachen (PHP, Java, SQL, Python parallel)	KI übernimmt Syntax-Umsetzung zuverlässig; Sprachgrenzen werden durchlässig	**Eine** Sprache gründlich verstehen (Lesen, Debuggen), Konzepte transferieren
Mechanisches Debugging von Typ-Fehlern, Library-Inkompatibilitäten	KI-Agenten lösen Versions- und Abhängigkeitskonflikte effizienter als Menschen	Konzeptuelles Debugging: **Warum** ist etwas falsch, nicht nur **wo**
Auswendiglernen von API-Signaturen und Sprachspezifika	Abrufbares Wissen, das KI in Millisekunden liefert	Lesen und Bewerten von Dokumentation
Boilerplate-Code schreiben (CRUD, Getter/Setter, Formularvalidierung)	Standardaufgaben, die KI am zuverlässigsten löst	Architekturentscheidungen treffen: **Welchen** Ansatz wähle ich und warum?
Implementierung von Standardalgorithmen aus dem Gedächtnis	Reproduktion ist KI-Kernkompetenz	Algorithmen **verstehen, vergleichen, bewerten** – z. B. Dijkstra nicht implementieren, sondern auf Graphenproblemen **anwenden**

6.3 Was neu hinzukommt

Neue Kompetenz	Beschreibung	Konkreter Unterrichtsbezug
Context Engineering	Systemprompts, XML-Strukturierung, Few-Shot-Beispiele, Constraints für präzise KI-Steuerung	Bereits als eigenständiges Seminarkapitel erarbeitet; direkt einsetzbar
KI-Output evaluieren	Sicherheit, Korrektheit, Wartbarkeit, Effizienz von KI-generiertem Code bewerten	Code-Review-Übungen: SuS erhalten KI-generierten Code und müssen Fehler finden
Problemzerlegung für Agenten	Ein komplexes Problem so aufteilen, dass KI-Agenten es in Teilschritten lösen können	Direkte Erweiterung des algorithmischen Denkens in Richtung Orchestrierung
ML/KI-Konzepte	Next Token Prediction, Embeddings, Entscheidungsbäume, Data Leakage, Overfitting	Ergänzt das klassische Curriculum um die Frage: Wie funktioniert das Werkzeug, das wir nutzen?
Ethik & Verantwortung	Wer trägt Verantwortung für KI-generierten Code? Bias, Halluzinationen, Urheberrecht	Fallstudien zu realen Vorfällen (z. B. gelöschte Datenbanken, Sicherheitslücken)

Aus der Hochschullehre: ML-Implementierung als Beispiel

Bei der Vermittlung von Machine Learning an Studierende zeigt sich das Problem exemplarisch: Stundenlange Kämpfe mit Python-Library-Inkompatibilitäten (Versionsnummern müssen exakt aufeinander abgestimmt sein) kosten wertvolle Lernzeit. Währenddessen übersieht man das fehlerhafte Konzept: Data Leakage zwischen Trainings- und Testdaten, Overfitting des Modells, fehlerhafte Parametrierung des Neuronalen Netzes. Syntax-Kleinkariertheit führt zu Fehlern in der Pipeline. Genau hier muss die Implementierungsarbeit durch KI-Assistenten abgenommen werden, damit der Fokus auf den konzeptuellen Fehlern liegt.

7 Handlungsempfehlung: Das Drei-Phasen-Modell

Die folgende Empfehlung richtet sich an Informatik-Lehrkräfte, die ihren Unterricht an die neue Realität anpassen wollen – unabhängig davon, wann der Bildungsplan nachzieht. Das Modell ist sofort umsetzbar und mit dem bestehenden Curriculum kompatibel.

Phase 1: Grounding – Verstehen ohne KI

Ziel: Schülerinnen und Schüler können Code **lesen, verstehen und bewerten**. Sie entwickeln ein belastbares mentales Modell davon, was ein Programm tut und warum.

Didaktisches Prinzip: Eine Sprache gründlich verstehen statt mehrere oberflächlich. Python bietet sich als Universalsprache an: niedrige Einstiegshürde, breite Anwendung, gute Übertragbarkeit der Konzepte.

Bewusster Verzicht auf KI in dieser Phase – nicht aus Technikfeindlichkeit, sondern aus pädagogischer Notwendigkeit. Das Bayerische Forschungsinstitut für Digitale Transformation (bidt) bestätigt: Grundlegende Programmierkonzepte können nur durch eigenes Tun (Learning by Doing) erworben werden. Der Extremfall wäre, dass mit KI gelöste Übungsblätter mittels KI korrigiert werden.

Thema (klassisch)	Fokus-Verschiebung	Konkretes Beispiel
Variablen, Datentypen, Kontrollstrukturen	Weniger Syntax-Drill, mehr **konzeptuelles Verständnis** (Was passiert im Speicher?)	Statt 20 Syntaxübungen: 5 gezielte Aufgaben + Tracing-Übungen, bei denen SuS Code **von Hand** durchlaufen
Algorithmen (Sortieren, Suchen)	Nicht auswendig implementieren, sondern **vergleichen, bewerten, anwenden**	SuS erhalten drei Sortieralgorithmen und bewerten: Welcher eignet sich für welches Szenario? Warum?
Datenstrukturen (Listen, Bäume, Graphen)	**Modellierung** vor Implementierung: Welche Struktur passt zum Problem?	SuS modellieren ein soziales Netzwerk als Graph und diskutieren: Welche Fragen kann man damit beantworten?
Automatentheorie, Formale Sprachen	**Bleibt zentral** – formale Modellierung ist genau das, was KI nicht kann	Endliche Automaten für Eingabevalidierung; Grammatiken zum Verständnis von Parsing und Sprachverarbeitung
Relationale Datenbanken	ER-Modellierung und Normalisierung als **Denkwerkzeug**, weniger SQL-Syntax-Drill	SuS entwerfen ein Datenbankschema für ein reales Problem; SQL-Abfragen werden später mit KI-Unterstützung geschrieben

Phase 2: KI als Werkzeug – Arbeiten mit KI

Ziel: Schülerinnen und Schüler setzen KI-Tools produktiv ein, können deren Output kritisch bewerten und kennen die Grenzen.

Didaktisches Prinzip: Die KI ist ein **sehr schneller, aber unzuverlässiger Praktikant** – mit beeindruckender Geschwindigkeit, aber ohne Urteilsvermögen. Die Metapher stammt sinngemäß von Matt Welsh selbst und eignet sich hervorragend für den Unterricht.

Unterrichtsformat	Beschreibung	Bezug zu Phase 1
KI-Code-Review	SuS erhalten KI-generierten Code für ein bekanntes Problem und müssen Fehler, Sicherheitslücken und Stilprobleme identifizieren	Setzt Lesefähigkeit aus Phase 1 voraus
Inverses Programmieren	SuS beschreiben ein Problem in natürlicher Sprache, die KI generiert Code, SuS bewerten und verbessern	Verbindet Context Engineering mit Code-Verständnis
Konsistenz-Check	SuS stellen dieselbe Aufgabe dreimal unterschiedlich formuliert und vergleichen die KI-Outputs	Trainiert kritisches Denken und zeigt Grenzen der KI
ML-Pipeline mit KI-Support	Implementierung eines ML-Modells; KI übernimmt Library-Setup und Boilerplate; SuS fokussieren auf **konzeptuelle Fehler** (Data Leakage, Overfitting)	Befreit von Syntax-Kampf, ermöglicht tieferes Konzeptverständnis
Prompt-Duell	Zwei Gruppen versuchen, dasselbe Problem mit unterschiedlichen Prompts zu lösen; Vergleich der Ergebnisse	Motivierend, kompetitiv, zeigt den Einfluss präziser Formulierung

Phase 3: Architektur und Systemdenken – Die höhere Ebene

Ziel: Schülerinnen und Schüler können Systeme entwerfen, Entscheidungen abwägen und die Rolle des KI-Orchestrators übernehmen.

Didaktisches Prinzip: Hier wird der Unterricht zum Architektur-Seminar. Die Fragen lauten nicht mehr „Wie implementiere ich das?“, sondern „Warum entscheide ich mich für diesen Ansatz?“ und „Welche Konsequenzen hat diese Entscheidung in sechs Monaten?“

Unterrichtsformat	Beschreibung	Lernziel
Architektur-Werkstatt	SuS erhalten ein komplexes Problem (z. B. Schulverwaltungssystem) und entwerfen die Architektur: Datenmodell, Schnittstellen, Module. Implementierung übernimmt KI.	Systemdenken, Entwurfsentscheidungen, Trade-offs
Post-Mortem-Analyse	SuS analysieren reale Fälle von gescheiterten Vibe-Coding-Projekten (z. B. EnrichLead, gelöschte Datenbank) und erarbeiten: Was hätte man anders machen müssen?	Verantwortung, Sicherheitsdenken, Risikobewertung
KI-Agenten orchestrieren	SuS zerteilen ein Projekt in Teilaufgaben, delegieren an KI-Agenten und integrieren die Ergebnisse. Reflexion: Welche Zerlegung funktioniert, welche nicht?	Problemzerlegung, Qualitätskontrolle, „Agentic Engineering“ nach Karpathy
Ethik-Fallstudie	Wer trägt die Verantwortung, wenn KI-generierter Code Schaden anrichtet? Diskussion anhand realer Vorfälle.	Ethische Reflexion, Verantwortungsbewusstsein
Vergleichende Modellierung	Dasselbe Problem mit unterschiedlichen Paradigmen lösen: relational vs. dokumentenbasiert, iterativ vs. rekursiv, zentral vs. verteilt	Paradigmenverständnis, begründetes Entscheiden

8 Konkrete Unterrichtssequenzen

Die folgenden Sequenzen sind direkt einsetzbar und zeigen exemplarisch, wie das Drei-Phasen-Modell in bestehende Themenblöcke integriert werden kann.

8.1 Sequenz: „Der KI-Code-Detektiv“ (2 Unterrichtsstunden)

Einordnung: Phase 2 | Thema: Algorithmen & Datenstrukturen
Vorwissen: SuS kennen grundlegende Sortieralgorithmen und Datenstrukturen aus Phase 1.
Ziel: SuS können KI-generierten Code kritisch bewerten und typische KI-Fehler identifizieren.

Ablauf Stunde 1: Die Lehrkraft generiert vorab mit einem LLM drei Lösungen für dasselbe Problem (z. B. „Finde den kürzesten Weg in einem gewichteten Graphen“). Mindestens eine Lösung enthält einen subtilen Fehler. SuS arbeiten in Dreiergruppen und analysieren den Code nach dem Bewertungsraster:

Prüfkriterium	Leitfrage
Korrektheit	Liefert der Algorithmus für alle Eingaben das richtige Ergebnis? Gibt es Randfälle?
Effizienz	Welche Laufzeitkomplexität hat die Lösung? Gibt es unnötige Berechnungen?
Lesbarkeit	Sind Variablennamen sprechend? Ist die Struktur nachvollziehbar?
Sicherheit	Gibt es potenzielle Angriffsvektoren? Ungeprüfte Eingaben?
Wartbarkeit	Könnte man den Code in 6 Monaten noch verstehen und erweitern?

Ablauf Stunde 2: Gruppen präsentieren ihre Befunde. Gemeinsame Diskussion: Welche Fehler hätte man ohne Algorithmen-Wissen übersehen? Ergebnis: Die Klasse erstellt kollektiv eine „Checkliste für KI-Code-Review“.

8.2 Sequenz: „Vom ER-Modell zur KI-Datenbank“ (3 Unterrichtsstunden)

Einordnung: Phasen 1→2 | Thema: Relationale Datenbanken
Vorwissen: Grundlagen der Datenmodellierung. Ziel: SuS erkennen den Wert konzeptueller Modellierung als Kontrollinstanz für KI-generierte Implementierungen.

Stunde 1 (Phase 1, ohne KI): SuS entwerfen ein ER-Diagramm für ein Schulbibliothekssystem. Entitäten: Bücher, Autoren, Schüler, Ausleihen. Beziehungstypen und Kardinalitäten werden diskutiert. Anschließend: Normalisierung bis 3NF.

Stunde 2 (Phase 2, mit KI): SuS geben der KI den Auftrag: „Erstelle ein relationales Datenbankschema für ein Schulbibliothekssystem mit Büchern, Autoren, Schülern und Ausleihen.“ Die KI liefert ein Schema. SuS vergleichen es mit ihrem eigenen ER-Modell:

Prüfpunkt	Typische KI-Fehler
Normalisierung	KI packt Autornamen direkt in die Bücher-Tabelle (Redundanz, 1NF-Verletzung)
Beziehungstypen	m:n-Beziehung Buch↔Autor wird als 1:n modelliert (kein Zwischentabelle)
Datentypen	ISBN als Integer statt String; Datum als Text statt Date
Constraints	Fehlende Fremdschlüssel, fehlende NOT NULL-Constraints
Benennung	Inkonsistente Namenskonventionen (CamelCase und snake_case gemischt)

Stunde 3: SuS formulieren einen **verbesserten Prompt** auf Basis ihrer Erkenntnisse (Context Engineering!) und vergleichen das neue Ergebnis. Reflexion: Wie viel Domainwissen muss man mitbringen, um KI-Output bewerten zu können?

9 Zusammenfassung und Ausblick

Die richtige Frage lautet: **Welches Programmieren brauchen wir, und wozu?**
Die KI hat die Abstraktionsleiter um eine Stufe verlängert. Wie bei jeder vorherigen Stufe gilt:

> **Kernthese**
> Die darunterliegende Schicht muss man verstehen, aber man muss nicht seine gesamte Zeit dort verbringen. Wer Algorithmen versteht, muss nicht jede Schleife von Hand tippen. Wer Datenmodellierung beherrscht, muss nicht jede SQL-Query auswendig schreiben. Wer Automatentheorie kennt, kann formale Probleme präzise beschreiben – und genau das ist es, was KI-Agenten brauchen, um effektiv zu arbeiten.

Für die Unterrichtspraxis bedeutet das konkret:

Erstens: Die Phase des Grounding – das eigenhändige Programmieren ohne KI-Unterstützung – bleibt unverzichtbar. Nur wer Code lesen und verstehen kann, kann KI-Output bewerten. Das Stanford-Vertrauens-Paradox zeigt, was passiert, wenn diese Basis fehlt.

Zweitens: Die freiwerdende Zeit (durch weniger Syntax-Drill, weniger Sprachvielfalt, weniger mechanisches Debugging) sollte investiert werden in: vertiefte Algorithmen und Graphentheorie, ML- und KI-Konzepte, Architektur- und Systemdenken, sowie Context Engineering als neue Schlüsselkompetenz.

Drittens: Der Bildungsplan wird erfahrungsgemäß noch 3–4 Jahre brauchen, um nachzuziehen. In dieser Zeit bilden wir 2–3 Generationen zukünftiger Informatiker heran. Diese Schülerinnen und Schüler können nicht warten. Lehrkräfte, die jetzt handeln, verschaffen ihren Schülern einen entscheidenden Vorsprung.

10 Quellen und Referenzen

Karpathy, A. (2025): „There's a new kind of coding I call 'vibe coding'...". X/Twitter, 2. Februar 2025.
Karpathy, A. (2025): „I've never felt this much behind as a programmer." X/Twitter, Dezember 2025.
Karpathy, A. (2026): „Agentic Engineering". X/Twitter, 4. Februar 2026.
Roden, G. (2026): „Kommentar: Entwickler, wacht auf – oder verliert euren Job". Heise Online, 19. Januar 2026.
Welsh, M. (2023): „The End of Programming". Communications of the ACM, Januar 2023.
Welsh, M. (2023): „Large Language Models and The End of Programming". CS50 Tech Talk, Harvard University.
Ng, A. (2025): Zitiert in: „Programmieren lernen im Zeitalter der Künstlichen Intelligenz". Machine Learning Blog, März 2025.
TeachAI / Gesellschaft für Informatik (2025): „Guidance on the Future of Computer Science Education in an Age of AI". teachai.org/cs, Juli 2025.
Veracode (2025): GenAI Code Security Report 2025.
CodeRabbit (2025): Analyse von 10+ Millionen Pull Requests zu KI-generiertem Code.
Uplevel Research (2025): Studie zu Bugdichte in KI-unterstütztem Code.
Stanford University (2025): Studie zu Sicherheitsverhalten bei KI-gestützter Entwicklung.
Stack Overflow (2025): Developer Survey 2025.
Bayerisches Forschungsinstitut für Digitale Transformation – bidt (2025): Projekt pAIrProg.
OECD/PISA (2025): Aufnahme von Computational Thinking in das PISA-Testformat.
Darrow, K. (2026): „State of Vibecoding in Feb 2026". kristindarrow.com.
New Stack (2026): „Vibe Coding Could Cause Catastrophic Explosions in 2026". 20. Januar 2026.
New Stack (2026): „Vibe Coding is Passé. Karpathy Has a New Name." 10. Februar 2026.
CACM (2025): „Catching the Vibe of Vibe Coding". Communications of the ACM, Mai 2025.
Wang, X., Wan, F. & Dai, J. (2026): „Enhancing Computational Thinking through Coding Education". Frontiers in Psychology, 17.
Mills, K. A. et al. (2024): „Coding and Computational Thinking Across the Curriculum". Review of Educational Research.