Programmierung von CAx-Systemen

David Straub

Software Engineering Basics

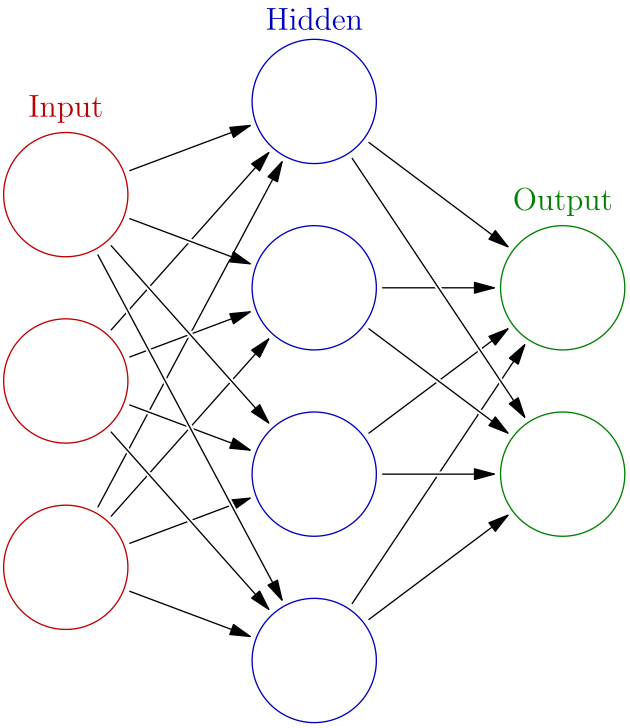
1. Versionsverwaltung mit Git
2. Unittests mit Pytest
3. Type Hints und statische Codeanalyse
4. **Agentische KI**

Überblick

1. Large Language Models (LLMs)
2. Wie funktioniert ein Coding-Agent?
3. Coding-Agent einrichten
4. Coding-Agenten in der Praxis

Large Language Models (LLMs)

Was ist ein Neuronales Netz?



Ein **neuronales Netz** ist eine Funktion, die aus Daten lernt.

Es besteht aus Schichten von Knoten (Neuronen). Jede Schicht transformiert ihre Eingabe durch **Matrixmultiplikationen und einfache nichtlineare Funktionen** – das wiederholt sich viele Male.

Die freien Parameter dieses Systems (die **Gewichte**) werden beim **Training** automatisch so eingestellt, dass das Netz auf gegebenen Beispielen die richtigen Ausgaben liefert.

Das Modell lernt **keine expliziten Regeln** – es lernt Muster direkt aus Daten.

Neuronale Netze: Anwendungsbeispiele

Eingabe	Ausgabe	Anwendung
Röntgenbild	Wahrscheinlichkeit „Tumor“	Medizinische Diagnose

Eingabe	Ausgabe	Anwendung
Sensordaten (Beschleunigung, Temperatur)	verbleibende Laufzeit bis Ausfall	Predictive Maintenance
Punktwolke (z. B. Laserscan eines Bauteils)	Bauteilkategorie	Automatische Klassifikation

Das Prinzip ist immer gleich: genug Beispielpaare (Eingabe → gewünschte Ausgabe) + Training = ein Modell, das auf neuen Eingaben verallgemeinert.

Generative Modelle

Die bisherigen Beispiele liefern alle eine Klasse oder einen Wert als Ausgabe.

Generative Modelle lernen stattdessen, neue Daten zu **erzeugen**, die den Trainingsdaten ähneln:

Trainiert auf	Erzeugt
Fotos von Gesichtern	neue, realistische Gesichter
3D-Modelle von Bauteilen	neue Geometrievorschläge
Musikstücke eines Stils	neue Kompositionen
Text in allen Varianten	neuen Text

Das Lernziel ändert sich: nicht „was ist das?“ sondern „wie geht das weiter?“ oder „was passt dazu?“.

LLMs sind generative Modelle – spezialisiert auf Text.

LLMs – ein Spezialfall neuronaler Netze

LLMs sind neuronale Netze, die auf **Text** spezialisiert sind.

Die Lernaufgabe beim Training ist denkbar einfach: **Sag das nächste Wort vorher.** Trainiert auf Milliarden von Seiten Text reicht das aus, um ein Modell zu erzeugen, das schreibt, erklärt, übersetzt und Code produziert.

Ein LLM-Aufruf ist probabilistisch: Ausgehend vom bisherigen Text (dem **Kontext**) liefert das Modell eine **Wahrscheinlichkeitsverteilung** über mögliche nächste Tokens (Wörter oder Wortfragmente). Das wahrscheinlichste wird gewählt – dann wiederholt sich der Vorgang.

```
Kontext:  "def berechne_querschnitt(radius:"
P(" float") = 0.73,  P(" int") = 0.12,  P(" str") = 0.01, ...
→ wähle " float"  →  neuer Kontext  →  nächstes Wort  →  ...
```

Was ein LLM kann und nicht kann

Kann	Kann nicht
Text zusammenfassen, erklären, übersetzen	Wirklich „verstehen“
Code schreiben, debuggen, umstrukturieren	Garantiert korrekte Ergebnisse liefern
Fragen beantworten (Trainingsdaten)	Aktuelle Ereignisse kennen (ohne Tools)
Kontext über viele Zeilen halten	Intern rechnen (zuverlässig)

Faustregel: LLMs sind sehr gute *Wahrscheinlichkeitsmaschinen* – kein Orakel, kein Compiler.

Meilensteine der jüngeren LLM-Entwicklung

Jahr	Ereignis
Nov 2022	**ChatGPT** – erstes LLM für die breite Öffentlichkeit (OpenAI GPT-3.5)
Feb 2023	**LLaMA** (Meta) – erstes leistungsfähiges open-weights Modell
März 2023	**GPT-4** – multimodal, deutlich stärker; **GitHub Copilot X** angekündigt
Juli 2023	**Code Interpreter** in ChatGPT – LLM führt Python-Code aus
März 2024	**Claude 3** (Anthropic), **Gemini 1.5** (Google) – Kontextfenster >1 Mio. Wörter
April 2024	**Llama 3** – open-weights Modelle erreichen kommerzielle Qualität
Okt 2024	**Cursor**, **Copilot Edits** – LLM-gestützte Code-Bearbeitung im Editor
Jan 2025	**DeepSeek R1** – open-weights Modell auf GPT-4-Niveau, zu einem Bruchteil der Trainingskosten

Jahr	Ereignis
Anfang 2025	**Claude 3.7 Sonnet** – extended thinking; komplexe Coding-Aufgaben end-to-end

Wie funktioniert ein Coding-Agent?

Schritt 1: Chat – str → str

Ein normaler LLM-Aufruf ist zustandslos:

```
messages = [
    {"role": "system",  "content": "Du bist ein hilfreicher Assistent."},
    {"role": "user",    "content": "Was ist 2 + 2?"},
]
reply = model.query(messages)   # → {"role": "assistant", "content": "4"}
```

Eingabe: Liste von Nachrichten. Ausgabe: eine neue Nachricht. Das war's.

Schritt 2: Tools – das LLM darf handeln

Das LLM kann in seiner Antwort **Werkzeugaufrufe** (tool calls) einbetten:

```
Aufgabe: "Ergänze Type Hints in rohr.py"

LLM antwortet:
  Ich lese zuerst die Datei.
  → Tool: bash(command="cat rohr.py")
```

Das Programm führt den Befehl aus und hängt das Ergebnis als neue Nachricht an.

Schritt 3: Die Agentenschleife

```
def run(task: str) → str:
    messages = [system_prompt, task]
    while True:
        reply   = model.query(messages)     # LLM antwortet + plant nächsten Schritt
        outputs = env.execute(reply.actions) # Tool-Aufrufe ausführen
        messages += [reply, outputs]         # Ergebnis zurück in den Kontext
        if reply.is_done():
            break
    return reply.submission
```

Beispieldurchlauf

Aufgabe: *„Ergänze Type Hints in rohr.py"*

Schritt	LLM	Tool-Ergebnis
1	`cat rohr.py`	`def rohr(r_aussen, wandstaerke, laenge):`
2	`sed -i 's/def rohr(r_aussen, wandstaerke, laenge)/def rohr(r_aussen: float, wandstaerke: float, laenge: float) → Solid/' rohr.py`	Exit-Code 0
3	`mypy rohr.py`	`Success: no issues found`
4	`echo COMPLETE_TASK_…`	→ fertig

Jeder Schritt = ein LLM-Aufruf. Das Ergebnis wird als neue Nachricht an den Kontext angehängt.

Grenzen und Risiken

Problem	Konkret
Halluzination	Agent erfindet nicht existierende Funktionen / APIs
Endlosschleife	Agent dreht sich im Kreis, ohne Fortschritt
Kontextverlust	Bei langen Aufgaben „vergisst" das LLM frühere Schritte
Kosten	Viele Tool-Aufrufe = viele LLM-Anfragen = hohe API-Kosten

Konsequenz: Agenten brauchen klare Aufgabenstellung, Versionskontrolle (Git!) und menschliche Überprüfung.

Coding-Agent einrichten

Coding-Agenten in der IDE

Coding-Agenten gibt es als Kommandozeilen-Tools, aber auch als Plugins für IDEs (z. B. VS Code). Bekannte Beispiele:

Tool	Hersteller	IDE	Modelle
GitHub Copilot	Microsoft / GitHub	VS Code	GPT-4o, Claude 3.5/3.7, Gemini 1.5 u. a.
Claude Code for VS Code	Anthropic	VS Code	Claude 3.5/3.7 Sonnet, Claude Opus
Cursor	Anysphere	Cursor (eigene IDE)	GPT-4o, Claude 3.5/3.7, Gemini u. a.
Codex	OpenAI	VS Code	GPT-4o, o3

Einrichten: Cline + Mistral Devstral

Cline – Open-Source VS Code Extension für agentisches Coding **Mistral Devstral** – Coding-Modell von Mistral AI, in Europa entwickelt und gehostet

1. VS Code Extension *Cline* installieren
2. Mistral als Provider auswählen und API-Key erstellen

Coding-Agenten in der Praxis

Tokens und Kosten

LLMs verarbeiten Text nicht zeichenweise, sondern in **Tokens** – Wörtern oder Wortfragmenten. Ungefähr 1 000 Tokens entsprechen ~750 Wörtern.

API-Kosten werden pro Token berechnet – getrennt für Eingabe und Ausgabe:

```
Eingabe (Kontext):   alle bisherigen Nachrichten + gelesene Dateien
Ausgabe (Antwort):   LLM-Text + Tool-Aufrufe
```

Konsequenz: Lange Kontexte (viele Dateien, langer Chatverlauf) kosten mehr.
Unnötige Dateien im Kontext vermeiden – Qualität und Kosten hängen zusammen.

Plan-Modus vs. Agenten-Modus

Die meisten Tools bieten zwei Modi:

	Plan-Modus	Agenten-Modus
Was passiert	LLM entwirft einen Plan, führt nichts aus	LLM plant und führt sofort aus
Kontrolle	Mensch bestätigt vor jeder Aktion	Agent arbeitet autonom
Wann sinnvoll	Unbekannte Codebase, riskante Änderungen	Klar definierte, begrenzte Aufgaben

Faustregel: Erst Plan-Modus – dann im Agenten-Modus ausführen lassen.

Kontext gezielt steuern

Der Agent sieht nur, was im Kontext steht. Zu wenig Kontext → schlechte Ergebnisse. Zu viel → hohe Kosten und Ablenkung.

Gezielt Dateien einbinden: Nur die Dateien öffnen oder nennen, die für die Aufgabe relevant sind.

Aufgaben klein schneiden: Eine Aufgabe pro Session – nicht „refactore das gesamte Projekt“.

Chatverlauf zurücksetzen: Langer Verlauf erhöht Kosten und kann das Modell verwirren. Neue Aufgabe = neue Session.

AGENTS.md – Instruktionen für den Agenten

Offenes Format (Linux Foundation) für projektspezifische Agentenanweisungen – wird von Cline, Claude Code, Copilot, Cursor, Codex u. a. automatisch gelesen.

```
# AGENTS.md
## Setup
- Tests ausführen: `pytest tests/`

## Konventionen
- Type Hints für alle Funktionen
- Kommentare auf Deutsch
```

Kritische Einschränkungen zusätzlich in der Aufgabe wiederholen.

Halluzinationen bei spezialisierten Bibliotheken

LLMs kennen `numpy` oder `pandas` sehr gut – `build123d` oder `cadquery` sind weniger verbreitet. Die Folge: erfundene Methoden, falsche Signaturen, veraltete API.

```
# Agent schreibt – existiert nicht:
koerper.fillet_edges(radius=2)

# Richtig:
fillets(koerper, radius=2, edge_list=koerper.edges())
```

Gegenmaßnahmen: `AGENTS.md` mit API-Hinweisen befüllen, Type Checker (Pyright/Pylance) nach jeder Änderung prüfen lassen, Ergebnis visuell kontrollieren.

Dos & Don'ts

Konventionen

- `AGENTS.md` mit Hinweisen zur API befüllen (oder von dort auf eine separate Datei verweisen)

Type Hints

- Konventionen zum Type Checker (`pyright`) angeben und nach jeder Änderung ausführen lassen
- `cast` und `# type: ignore` akzeptieren oder übersehen

Tests

- Unittests erzeugen lassen (`.is_valid`, `.volume`, …) und ausführen lassen
- Unittest blind akzeptieren – Gefahr des „Cheating“!

Dos & Don’ts

Plan

- Zu umfangreiche Aufgabenstellung ohne Plan-Modus
- Architektur-Tradeoffs analysieren lassen

Analyse

- Änderungen erklären lassen
- Umständlich erscheinende Lösungen hinterfragen (*„Is this a hack?“ „Honestly, yes.“*)
- Übertrieben ausführliche Code-Kommentare übernehmen (gut geschriebener Code sollte selbsterklärend sein)
- Visuelle Überprüfung durchführen!