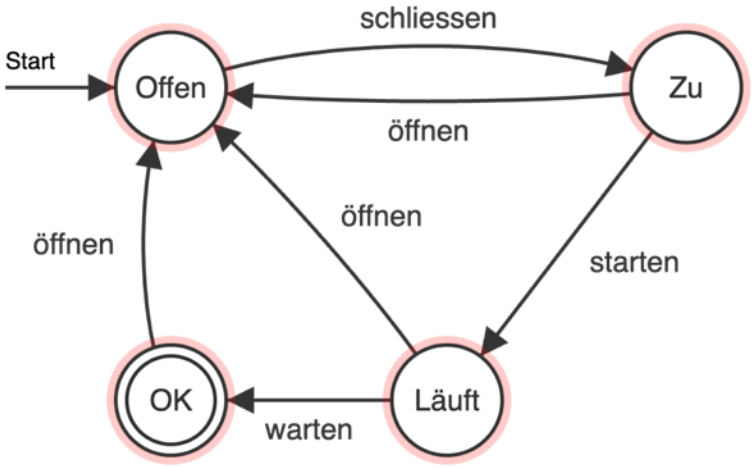
Zustandsautomaten als Modell für LangGraph-Agenten

Von formalen Grundlagen zur praktischen Implementierung

Grundlagen-Dokument
DQR5 2025/2026



Autor: Thomas Jörg. LFT KI / Dozent KI IHK, Intel

Table of Contents

1 Einführung

Wenn wir einen KI-Agenten bauen, der Werkzeuge benutzt, Entscheidungen trifft und in Schleifen arbeitet, dann bauen wir — ob bewusst oder nicht — einen Zustandsautomaten.

LangGraph, das Framework für agentische Workflows aus dem LangChain-Ökosystem, macht dieses Prinzip explizit: Jeder Agent wird als Graph definiert, mit Knoten, Kanten, einem Startpunkt und einem oder mehreren Endpunkten.

Die Automatentheorie liefert uns dafür drei Dinge:

- eine Notation (Zustandsdiagramme, die jeder lesen kann),
- ein Vokabular (präzise Begriffe für das, was passiert) und eine
- Denkdisziplin (erst der Graph, dann der Code).

> *Leitgedanke*
> Das Ziel ist, bei jedem Agenten sofort sagen zu können: „Das sind die Zustände, das sind die Übergänge, und hier fließen die Daten.“ — Erst zeichnen, dann implementieren.

Dieses Dokument baut das Verständnis in zwei Teilen auf:

Teil I
legt die formalen Grundlagen: Was ist ein endlicher Automat? Was unterscheidet Moore- von Mealy-Automaten? Was ist ein erweiterter Zustandsautomat?

Teil II
überträgt diese Konzepte auf LangGraph und zeigt, wie jedes theoretische Konzept eine direkte Entsprechung im Code hat.
Die Automatentheorie ist dabei Mittel zum Zweck — ein Denkwerkzeug, ähnlich wie man die abstrakte Datenstruktur des Binärbaums nutzt, um Decision Trees zu verstehen, oder den kNN-Algorithmus, um Similarity Search in Vektordatenbanken einzuordnen.

Teil I: Formale Grundlagen

2. Der deterministische endliche Automat (DEA)

Ein deterministischer endlicher Automat (DEA, englisch: DFA — Deterministic Finite Automaton) ist das einfachste formale Modell für zustandsbasierte Systeme. Er besteht aus fünf Komponenten:

Zustandsmenge	Q	Endliche Menge aller möglichen Zustände
Eingabealphabet	Σ	Endliche Menge der Eingabesymbole
Startzustand	q_0	Der Zustand, in dem der Automat beginnt
Endzustände	$F \subseteq Q$	Menge der akzeptierenden Zustände
Übergangsfunktion	$\delta: Q \times \Sigma \rightarrow Q$	Bestimmt den Folgezustand

Formal schreibt man einen DEA als 5-Tupel: $M = (Q, \Sigma, \delta, q_0, F)$

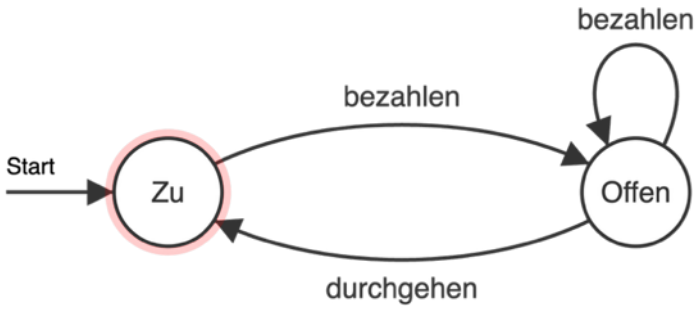
Beispiel 1: Drehkreuz (Turnstile)

Ein Drehkreuz an einer U-Bahn-Station hat zwei Zustände: gesperrt und entsperrt. Es reagiert auf zwei Eingaben: Münze einwerfen und Durchgehen.

Aktueller Zustand	Eingabe	Nächster Zustand
Zu	bezahlen	Offen
Offen	durchgehen	Zu
offen	bezahlen	Offen

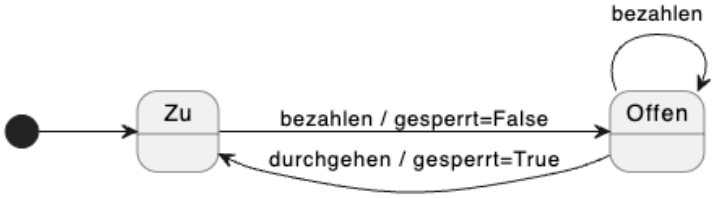
Formal:
Q = {Zu, Offen},
Σ = {bezahlen, durchgehen},
q_0 = Zu
F = {}

(kein akzeptierender Zustand, das Drehkreuz läuft endlos)
Im Zustandsdiagramm ist jeder Zustand ein Kreis, jeder Übergang ein beschrifteter Pfeil.
Der Startzustand wird durch einen eingehenden Pfeil markiert, Endzustände durch einen Doppelkreis.



δ	**bezahlen**	**durchgehen**
Zu	Offen	-
Offen	Offen	Zu

In der **Harel-Notation** (der gängigen UML-Notation) werden Zustände als abgerundete Rechtecke, Outputs innerhalb der Zustände mit 'Entry-Actions' bezeichnet und Outputs auf den Übergängen als 'Transition-Actions".



Das Zustandsdiagramm ist die visuelle Darstellung der Übergangstabelle. Beides enthält dieselbe Information — das Diagramm macht die Struktur sichtbar, die Tabelle macht sie präzise.

Langgraph Beispiel 1: das Drehkreuz als Langgraph-Objekt

Wichtiger Hinweis: Die Langgraph-Beispiele bitte erst nach dem Durcharbeiten von Teil II anschauen, da erst dort die Langgraph-Synthax eingeführt wird. Der Code steht hier, weil er inhaltlich zu den obigen Automaten gehört. Didaktisch ist er hier eigentlich noch viel zu früh.

```python
class DrehkreuzState(TypedDict):
    current_state: str
    input_sequence: list[str]
    position: int
    gesperrt: bool
    # Hinweis: 'input_sequence' ist ein Workaround, um einen Stream von Eingaben
    # zu simulieren. Es ist also ein didaktischer Hilfsgriff; die Input-Sequenz
    # dient hier nur als 'Prothese' um das formale Verhalten eines Automaten abzubilden

def drehkreuz_zu(state: DrehkreuzState) -> dict:
    pos = state["position"]
    eingabe = state["input_sequence"][pos]
    result = {"position": pos + 1}

    if eingabe == "bezahlen":
        result["current_state"] = "Offen"
        result["gesperrt"] = False
    return result


def drehkreuz_offen(state: DrehkreuzState) -> dict:
    pos = state["position"]
    eingabe = state["input_sequence"][pos]
    result = {"position": pos + 1}

    if eingabe == "durchgehen":
        result["current_state"] = "Zu"
        result["gesperrt"] = True
    return result


def drehK_router(state: DrehkreuzState) -> str:
    # ACHTUNG: Das Drehkreuz hat keinen Endzustand – es läuft
    # konzeptionell endlos. END beendet hier nur die Simulation,
    # weil die Eingabesequenz erschöpft ist.
    if state["position"] >= len(state["input_sequence"]):
        return END
    return state["current_state"]


def build_drehkreuz():
    graph = StateGraph(DrehkreuzState)
    graph.add_node("Zu", drehkreuz_zu)
    graph.add_node("Offen", drehkreuz_offen)
    graph.set_entry_point("Zu")
    graph.add_conditional_edges("Zu", drehK_router, {"Zu": "Zu", "Offen": "Offen", END: END})
    graph.add_conditional_edges("Offen", drehK_router, {"Zu": "Zu", "Offen": "Offen", END: END})
    # HINWEIS: In modernen LangGraph-Versionen kann das Dictionary weggelassen werden,
    # wenn der Router exakt den Knotennamen zurückgibt.
    return graph.compile()
```

Merkregel:
In Produktionscode gilt: Kritische Zustandsänderungen gehören auf die Transition, nicht in die Entry-Action des Zielknotens — weil der Zielknoten vielleicht nie läuft.

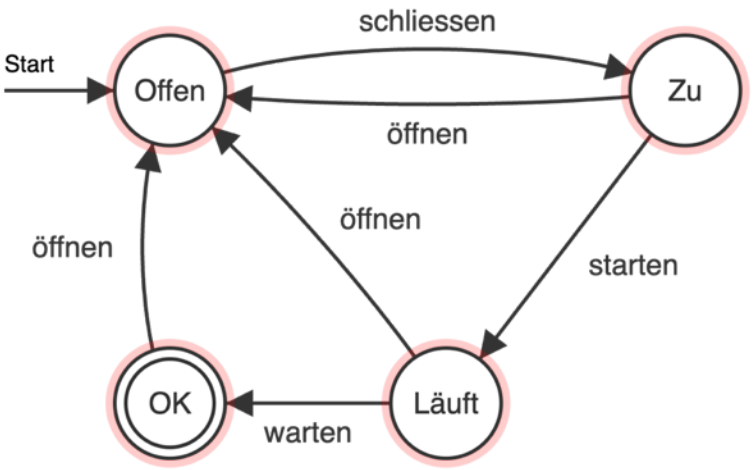
Beispiel 2: Mikrowelle als DEA

Eine Mikrowelle ist offen im Grundzustand, dann schließt man sie, startet und wartet, bis sie fertig ist. Macht man zwischendrin auf, dann geht sie wieder in den Grundzustand. Wenn die Wartezeit vorbei ist, geht sie in einen Endzustand, der hier mit ‚OK' benannt ist.

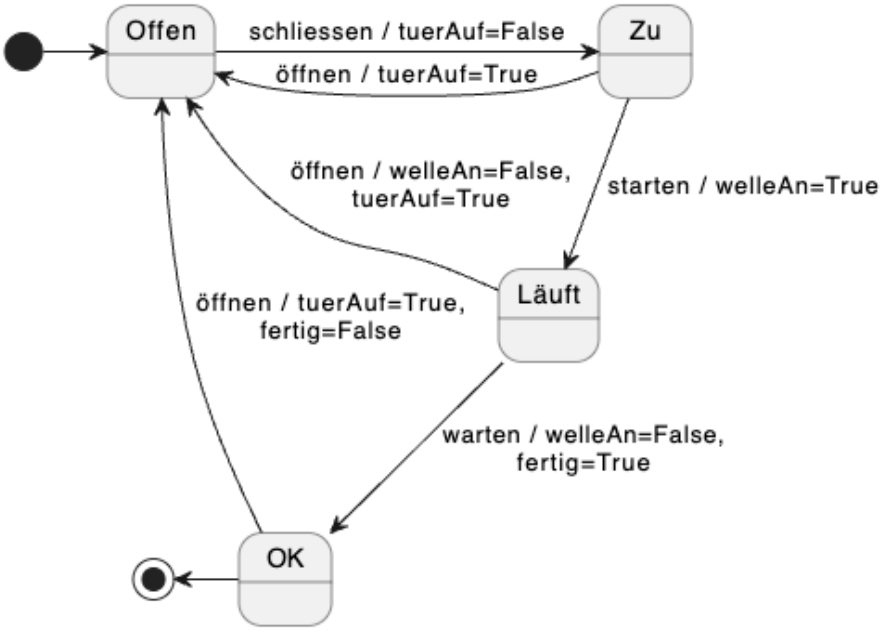
Aktueller Zustand	Eingabe	Nächster Zustand
Offen	schliessen	Zu
OK	öffnen	Offen
Läuft	öffnen	Offen
Läuft	warten	OK
Zu	öffnen	Offen
Zu	starten	Läuft

Formal:
Q = {Offen, Zu, Läuft, OK}
Σ = {schliessen, öffnen, starten, warten}
q_0 = Offen
F = {OK}

Achtung:
in der theoretischen Informatik ist ein Endzustand ein Merkmal eines Akzeptors; hier wird aber ein real existierender Prozess modelliert, ähnlich einem Use-case



δ	schlies	öffne	starten	warte
Offen	Zu			
OK		Offen		
Läuft		Offen		OK
Zu		Offen	Läuft	



Langgraph Beispiel 2: die Mikrowelle als Langgraph-Objekt

```
class MikroWState(TypedDict):
    aktState: str
    inputSeq: list[str]
    position: int
    tuerAuf: bool
    welleAn: bool
    fertig: bool


def mikrowelle_offen(state: MikroWState) ->
dict:
    pos = state["position"]
    eingabe = state["inputSeq"][pos]
    result = {"position": pos + 1}

    if eingabe == "schliessen":
        result["aktState"] = "Zu"
        result["tuerAuf"] = False
    return result


def mikrowelle_zu(state: MikroWState) ->
dict:
    pos = state["position"]
    eingabe = state["inputSeq"][pos]
    result = {"position": pos + 1}

    if eingabe == "öffnen":
        result["aktState"] = "Offen"
        result["tuerAuf"] = True
    elif eingabe == "starten":
        result["aktState"] = "Läuft"
        result["welleAn"] = True
    return result
```

```
def mikrowelle_laeuft(state: MikroWState) ->
dict:
    pos = state["position"]
    eingabe = state["inputSeq"][pos]
    result = {"position": pos + 1}

    if eingabe == "warten":
        result["aktState "] = "OK"
        result["welleAn"] = False
        result["fertig"] = True
    elif eingabe == "öffnen":
        result["currentState"] = "Offen"
        result["welleAn"] = False
        result["tuerAuf"] = True
    return result

def mikrowelle_ok(state: MikroWState) -> dict:
    pos = state["position"]
    eingabe = state["inputSeq"][pos]
    result = {"position": pos + 1}

    if eingabe == "öffnen":
        result["aktState"] = "Offen"
        result["tuerAuf"] = True
        result["fertig"] = False
    return result

def mikroW_router(state: MikroWState) -> str:
    if state["position"] >=
len(state["inputSeq"]):
        return END
    return state["aktState"]
```

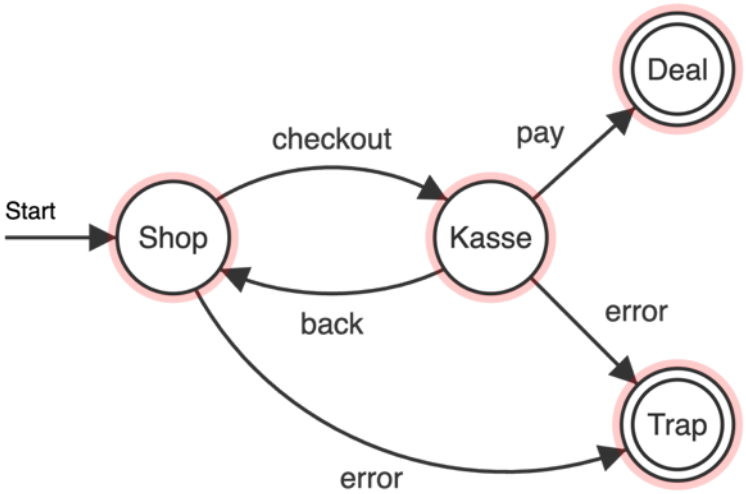
```
def build_mikrowelle():
    graph = StateGraph(MikroWState)
    graph.add_node("Offen", mikrowelle_offen)
    graph.add_node("Zu", mikrowelle_zu)
    graph.add_node("Läuft", mikrowelle_laeuft)
    graph.add_node("OK", mikrowelle_ok)
    graph.set_entry_point("Offen")
    for node in ["Offen", "Zu", "Läuft", "OK"]:graph.add_conditional_edges(node, mikroW_router,
            {"Offen": "Offen", "Zu": "Zu", "Läuft": "Läuft", "OK": "OK", END: END})
    return graph.compile()
```

Beispiel 3: Onlineshop als DEA

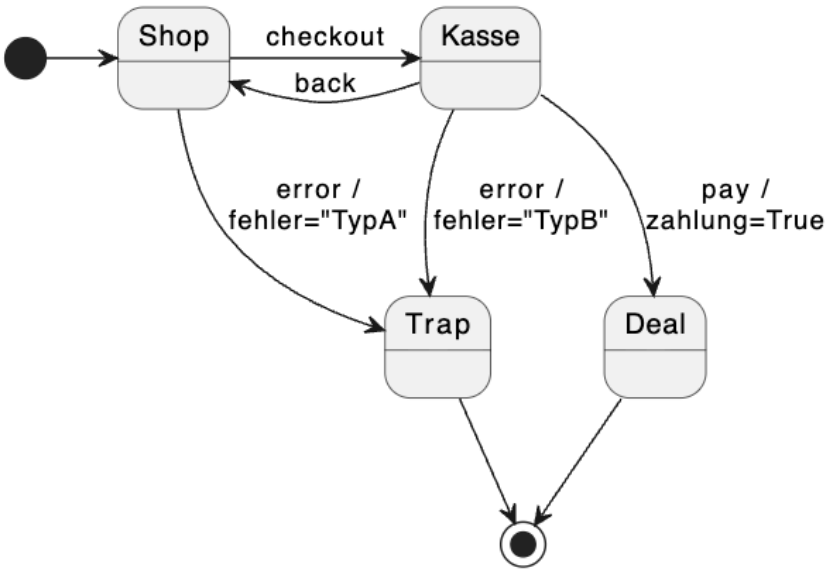
Hier wird ein Bezahlvorgang abgebildet. Vom Shop kommt man über den Checkout zur Kasse. Dort kann man entweder bezahlen oder es geschieht ein Fehler (z.B. wenn die Kreditkarte nicht erkannt wird oder der User es sich anders überlegt usw.)

Aktueller Zustand	Eingabe	Nächster Zustand
Shop	error	Trap
Shop	checkout	Kasse
Kasse	back	Shop
Kasse	pay	Deal
Kasse	error	Trap
Deal	-	-
Trap	-	-

Formal:
Q = {Shop, Kasse, Deal, Trap }
Σ = {error,checkout,back,pay}
q_0 = Shop
F = {Deal,Trap}



δ	pay	error	back	checkout
Shop		Trap		Kasse
Kasse	Deal	Trap	Shop	
Deal				
Trap				



Langgraph Beispiel 3: der Online-Shop als Langgraph-Objekt

```
class OnlineshopState(TypedDict):
    current_state: str
    input_sequence: list[str]
    position: int
    zahlung_gebucht: bool
    fehler: str | None


def shop_node(state: OnlineshopState) -> dict:
    pos = state["position"]
    eingabe = state["input_sequence"][pos]
    result = {"position": pos + 1}

    if eingabe == "checkout":
        result["current_state"] = "Kasse"
    elif eingabe == "error":
        result["current_state"] = "Trap"
        result["fehler"] = "Verbindungsfehler"
    return result


def kasse_node(state: OnlineshopState) -> dict:
    pos = state["position"]
    eingabe = state["input_sequence"][pos]
    result = {"position": pos + 1}

    if eingabe == "pay":
        result["current_state"] = "Deal"
        result["zahlung_gebucht"] = True
    elif eingabe == "error":
        result["current_state"] = "Trap"
        result["fehler"] = "Zahlungsfehler"
    elif eingabe == "back":
        result["current_state"] = "Shop"
    return result


def onlineshop_router(state: OnlineshopState) -> str:
    if state["current_state"] in ("Deal", "Trap"):
        return END
    if state["position"] >= len(state["input_sequence"]):
        return END
    return state["current_state"]


def build_onlineshop():
    graph = StateGraph(OnlineshopState)
    graph.add_node("Shop", shop_node)
    graph.add_node("Kasse", kasse_node)
    graph.set_entry_point("Shop")

    graph.add_conditional_edges(
        "Shop", onlineshop_router,{"Shop": "Shop", "Kasse": "Kasse", END: END}
    )

    graph.add_conditional_edges(
        "Kasse", onlineshop_router,{"Shop": "Shop", "Kasse": "Kasse", END: END}
    )
    return graph.compile()
```

4 Der Moore-Automat

Ein Moore-Automat erweitert den DEA um Ausgaben. Die zentrale Eigenschaft: Die Ausgabe hängt ausschließlich vom aktuellen Zustand ab — nicht von der Eingabe, die den Übergang ausgelöst hat. Formal wird ein Moore-Automat als 6-Tupel definiert: $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$. Neu gegenüber dem DEA sind:

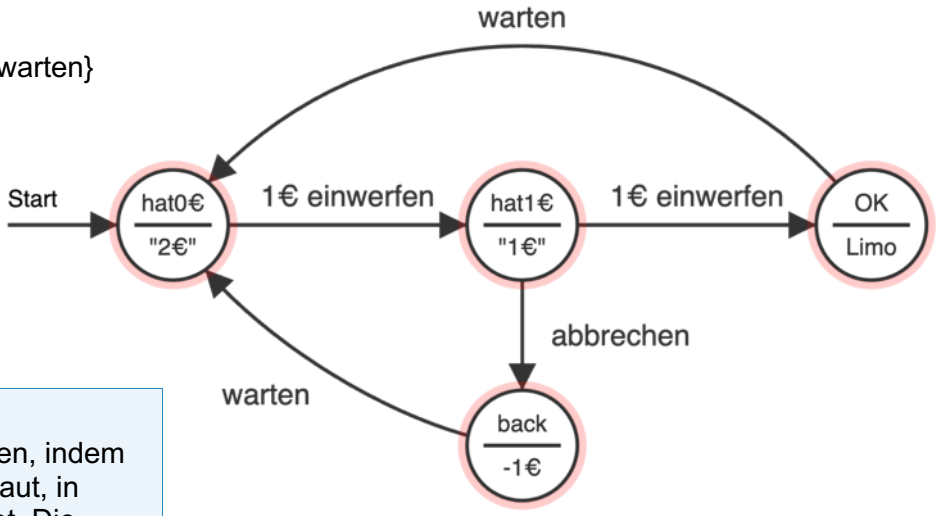
Komponente	Symbol	Bedeutung
Ausgabealphabet	Γ	Endliche Menge möglicher Ausgaben
Ausgabefunktion	$\lambda: Q \rightarrow \Gamma$	Ordnet jedem Zustand eine Ausgabe zu

Beispiel: Getränkeautomat (vereinfacht)
Ein vereinfachter Getränkeautomat akzeptiert 1€-Münzen. Ein Getränk kostet 2€. Der Automat hat drei Zustände, und jeder Zustand hat eine feste Ausgabe — unabhängig davon, welche Eingabe gerade kommt.

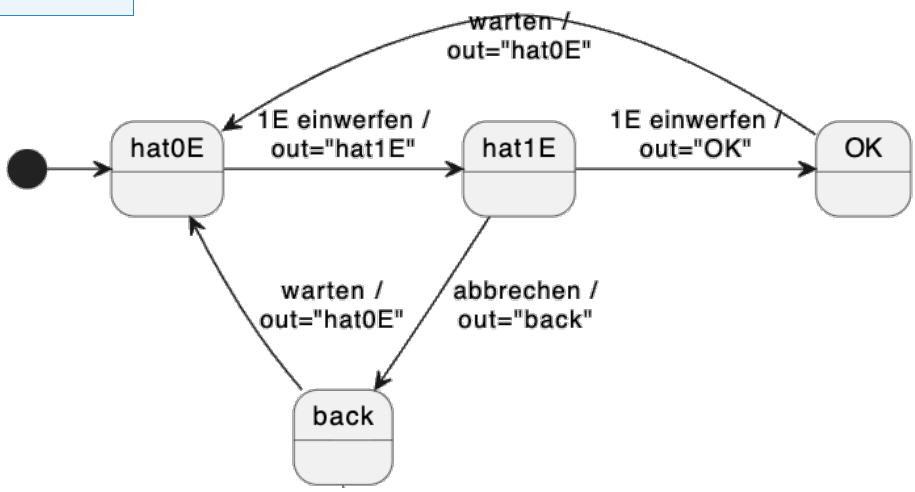
Zustand	Ausgabe (λ)	Eingabe	Nächster Zustand
hat0€ (q_0)	Anzeige: „2€“	1€ einwerfen	hat1€
hat1€	Anzeige: „1€“	1€ einwerfen	OK
hat1€	Anzeige: „1€“	abbrechen	Back
OK	Limo	warten	hat0€
back	1€ zurückgeben	warten	hat0€

Formal:

Q = {hat0€, hat1€, OK, back }
Σ = {1€ einwerfen, abbrechen, warten}
q_0 = hat0€
Γ = { "2€","1€",Limo,-1€}



Schlüsseleigenschaft Moore:
Man kann die Ausgabe ablesen, indem man nur auf den Zustand schaut, in dem der Automat sich befindet. Die Ausgabe steht „im Knoten“, nicht „an der Kante“.



Langgraph Beispiel 4: der Moor-Getränkeautomat als Langgraph-Objekt

```python
class DrinkMoorState(TypedDict):
    current_state: str
    inputSeq: list[str]
    position: int
    guthaben: int
    display: str
    outputs: Annotated[list[str],
operator.add]

MOOREOUT = {
    "hat0€": '"2€"',
    "hat1€": '"1€"',
    "OK":    "Limo",
    "back":  "-1€",
}

def moore_hat0(state: DrinkMoorState) -> dict:
    pos = state["position"]
    eingabe = state["inputSeq"][pos]
    result = {"position": pos + 1}

    if eingabe == "1€ einwerfen":
        ziel = "hat1€"
        result["current_state"] = ziel
        result["guthaben"] = 100
        result["display"] = MOOREOUT[ziel]
        result["outputs"] = [MOOREOUT[ziel]]
    return result

def moore_ok(state: DrinkMoorState) -> dict:
    pos = state["position"]
    eingabe = state["inputSeq"][pos]
    result = {"position": pos + 1}

    if eingabe == "warten":
        ziel = "hat0€"
        result["current_state"] = ziel
        result["guthaben"] = 0
        result["display"] = MOOREOUT[ziel]
        result["outputs"] = [MOOREOUT[ziel]]
    return result
```

```python
def moore_hat1(state: DrinkMoorState) -> dict:
    pos = state["position"]
    eingabe = state["inputSeq"][pos]
    result = {"position": pos + 1}

    if eingabe == "1€ einwerfen":
        ziel = "OK"
    elif eingabe == "abbrechen":
        ziel = "back"
    else:
        return result

    result["current_state"] = ziel
    result["guthaben"] = 200 if ziel == "OK"
else 0
    result["display"] = MOOREOUT[ziel]
    result["outputs"] = [MOOREOUT[ziel]]
    return result

def moore_back(state: DrinkMoorState) -> dict:
    pos = state["position"]
    eingabe = state["inputSeq"][pos]
    result = {"position": pos + 1}

    if eingabe == "warten":
        ziel = "hat0€"
        result["current_state"] = ziel
        result["guthaben"] = 0
        result["display"] = MOOREOUT[ziel]
        result["outputs"] = [MOOREOUT[ziel]]
    return result

def mooreRouter(state: DrinkMoorState) -> str:
    if state["position"] >=
len(state["inputSeq"]):
        return END
    return state["current_state"]

# Implementierungshinweis: Da LangGraph den
# Zielknoten nicht garantiert ausführt, setzen
# wir die  Ausgabe als Transition-Action. Das
# MOOREOUT-Dict stellt sicher, dass die zum
# Zielzustand gehörige Ausgabe verwendet wird.
```

```python
def build_getraenkeautomat_moore():
    graph = StateGraph(DrinkMoorState)
    graph.add_node("hat0€", moore_hat0)
    graph.add_node("hat1€", moore_hat1)
    graph.add_node("OK", moore_ok)
    graph.add_node("back", moore_back)
    graph.set_entry_point("hat0€")
    for node in ["hat0€", "hat1€", "OK", "back"]:
             graph.add_conditional_edges(node, mooreRouter,
            {"hat0€": "hat0€", "hat1€": "hat1€", "OK": "OK", "back": "back", END: END})
    return graph.compile()
```

5 Der Mealy-Automat

Der Mealy-Automat ist die Alternative zum Moore-Automaten. Der Unterschied: Die Ausgabe hängt nicht nur vom aktuellen Zustand ab, sondern auch von der aktuellen Eingabe. Die Ausgabe steht daher an der Kante, nicht im Knoten.

Formal: $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$, gleiche Struktur wie Moore, die Ausgabefunktion hat eine andere Signatur:

Typ	Ausgabefunktion	Ausgabe hängt ab von
Moore	$\lambda: Q \rightarrow \Gamma$	nur dem aktuellen Zustand
Mealy	$\lambda: Q \times \Sigma \rightarrow \Gamma$	Zustand UND Eingabe

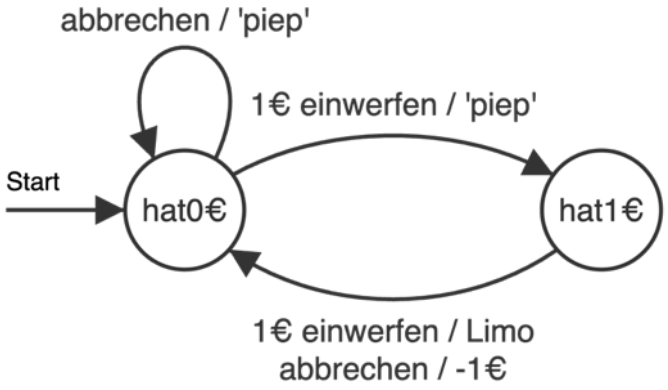
Beispiel: Mealy-Getränkeautomat

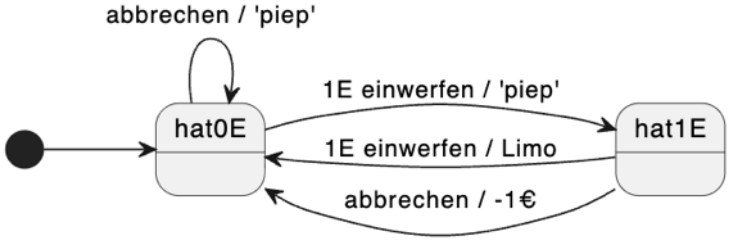
Ein vereinfachter Getränkeautomat – ähnlich dem Moore-Automaten oben - akzeptiert 1€-Münzen. Ein Getränk kostet 2€. Der Automat hat diesmal nur zwei Zustände. Die Ausgaben des Automaten erfolgen entlang der Übergänge. Dadurch können Zustände gespart werden.

Zustand	Eingabe	Ausgabe	Nächster Zustand
hat0€	1€ einwerfen	‘piep’	→ hat1€
hat0€	abbrechen	‘piep’	→ hat0€
hat1€	1€ einwerfen	Limo	→ hat0€
hat1€	abbrechen	-1€	→ hat0€

Im Zustandsdiagramm eines Mealy-Automaten werden die Kanten mit Eingabe/Ausgabe beschriftet, z.B. „abbrechen” / “piep“.

Formal:
Q = {hat0€, hat1€}
Σ = {1€ einwerfen, abbrechen}
q_0 = hat0€
Γ = { "piep", Limo,-1€}





Vergleich auf einen Blick
Moore: Ausgabe im Knoten („Zustand gibt X aus“). Mealy: Ausgabe an der Kante („bei Übergang wird X ausgegeben“). Jeder Moore- lässt sich in einen Mealy-Automaten umwandeln und umgekehrt.

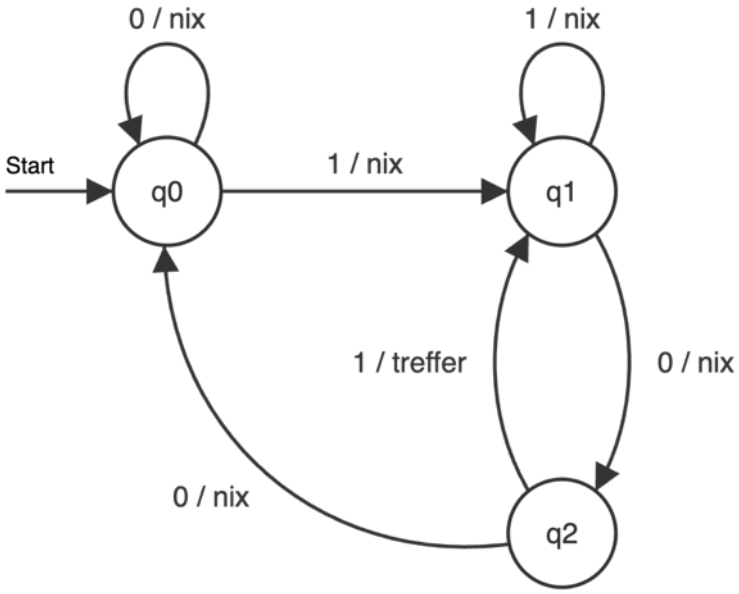
Langgraph Beispiel 5: der Mealy-Getränkeautomat als Langgraph-Objekt

```
class GetraenkeMealyState(TypedDict):
    current_state: str
    input_sequence: list[str]
    position: int
    guthaben_cent: int
    outputs: Annotated[list[str], operator.add]

MEALY_TRANSITIONS = {
    ("hat0€", "1€ einwerfen"): ("hat1€", "'piep'",  100),
    ("hat0€", "abbrechen"):    ("hat0€", "'piep'",    0),
    ("hat1€", "1€ einwerfen"): ("hat0€", "Limo",      0),
    ("hat1€", "abbrechen"):    ("hat0€", "-1€",       0),
}

def mealy_hat0(state: GetraenkeMealyState) -> dict:
    pos = state["position"]
    eingabe = state["input_sequence"][pos]
    ziel, ausgabe, guthaben = MEALY_TRANSITIONS[("hat0€", eingabe)]
    return {
        "current_state": ziel,
        "guthaben_cent": guthaben,
        "outputs": [ausgabe],
        "position": pos + 1,
    }

def mealy_hat1(state: GetraenkeMealyState) -> dict:
    pos = state["position"]
    eingabe = state["input_sequence"][pos]
    ziel, ausgabe, guthaben = MEALY_TRANSITIONS[("hat1€", eingabe)]
    return {
        "current_state": ziel,
        "guthaben_cent": guthaben,
        "outputs": [ausgabe],
        "position": pos + 1,
    }

def mealy_router(state: GetraenkeMealyState) -> str:
    if state["position"] >= len(state["input_sequence"]):
        return END
    return state["current_state"]

def build_getraenkeautomat_mealy():
    graph = StateGraph(GetraenkeMealyState)
    graph.add_node("hat0€", mealy_hat0)
    graph.add_node("hat1€", mealy_hat1)
    graph.set_entry_point("hat0€")
    for node in ["hat0€", "hat1€"]:graph.add_conditional_edges(node, mealy_router,
            {"hat0€": "hat0€", "hat1€": "hat1€", END: END})
    return graph.compile()
```

Beispiel 6: Mealy-Bitmustererkennung

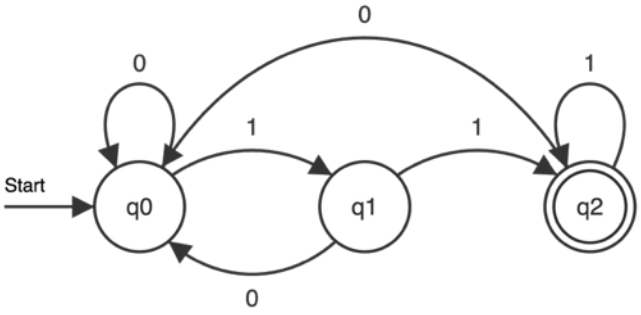
Man stelle sich einen Automaten vor, der einen endlosen Strom aus Nullen und Einsen liest. Seine Aufgabe: Er soll sofort "treffer" ausgeben, sobald die Sequenz "101" gelesen wurde. Das Komplexe daran sind überschneidende Muster. Wenn der Input 10101 lautet, muss der Automat zweimal auslösen (denn die Sequenz 101 taucht zweimal auf, sie teilen sich die mittlere 1).

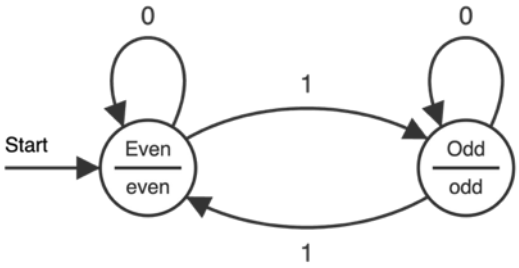
Formal:
Q = {q0, q1, q2}
Σ = {1, 0}
q_0 = q0
Γ = { "nix", “treffer”}

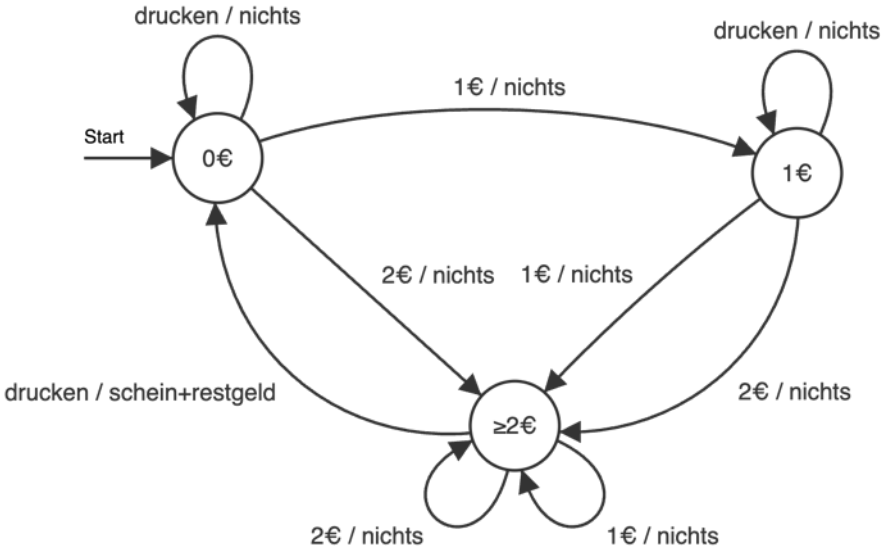


Übungsaufgaben:

Um welchen Automaten-Typ handelt sich?
Stelle den Formalismus auf?
Welche Eingaben führen wohin?
Welchem Zweck dient der Automat?







5 Vom Automaten zu Langgraph: der erweiterte Zustandsautomat EFSM

Sowohl DEA als auch Moore- und Mealy-Automaten haben eine wesentliche Einschränkung: Ihr Zustand ist nur die Antwort auf die Frage „In welchem Knoten bin ich gerade?“. Es gibt keinen Platz für zusätzliche Daten. In der Praxis reicht das nicht aus:

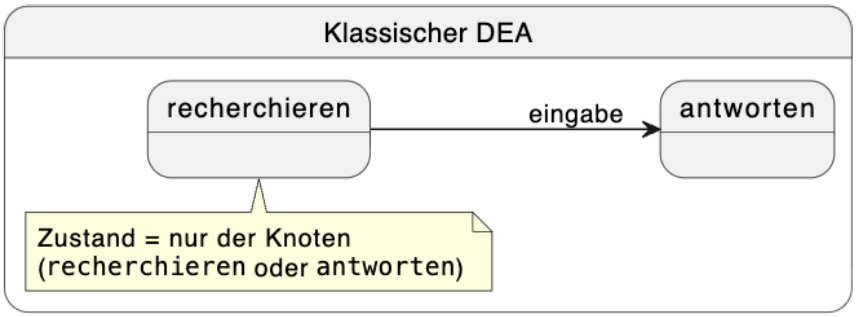
- Ein Agent, der eine Konversation führt, muss sich den bisherigen Verlauf merken.
- Ein Workflow, der Dokumente verarbeitet, muss Zwischenergebnisse speichern.
- Ein Retry-Mechanismus muss einen Zähler führen.

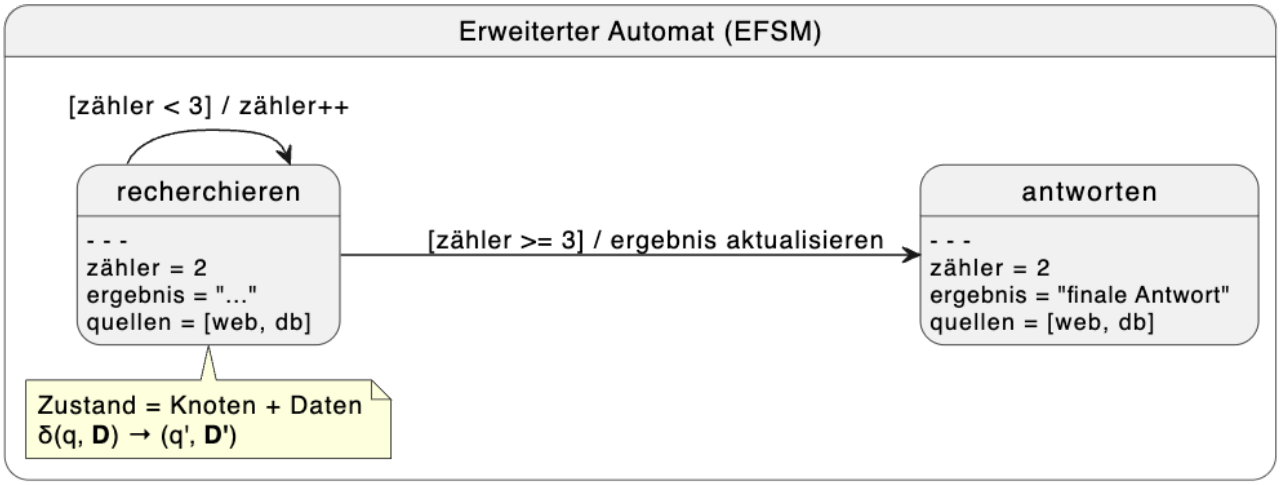
Der erweiterte Zustandsautomat (Extended State Machine, auch: Extended Finite State Machine, EFSM) löst dieses Problem, indem er den Zustand in zwei Teile aufspaltet:

Ebene	Bezeichnung	Frage	Beispiel
Kontrollzustand	Welcher Knoten?	In welcher Phase bin ich?	„recherchieren“
Datenzustand	Welche Daten?	Was weiß ich bisher?	{zähler: 2, ergebnis: „...“}

Der vollständige Zustand ist das Paar (Kontrollzustand, Datenzustand).

Die Übergangsfunktion kann nun beide Teile berücksichtigen: δ(q, D) → (q’, D’) — abhängig vom Knoten q UND den Daten D wird der nächste Knoten q’ bestimmt und die Daten zu D’ aktualisiert.





Die Brücke zu LangGraph
Der erweiterte Zustandsautomat ist das Modell, auf dem LangGraph aufbaut. Der Kontrollzustand ist der aktuelle Knoten im Graphen. Der Datenzustand ist das State-Objekt (TypedDict), das durch den Graphen wandert.

Teil II: Von der Theorie zu LangGraph

6 Konzept 1: Der Graph als Kontrollfluss

Ein Agent durchläuft eine Folge von Stationen. Jede Station erledigt eine Aufgabe. Der Weg von Station zu Station ist festgelegt. In der Automatentheorie sind die Stationen Zustände und die Wege Übergänge.

Automatentheorie	LangGraph-Code	Bedeutung
Zustand q	add_node("name", fn)	Definiert eine Station
Startzustand q_0	add_edge(START, "name")	Markiert den Einstieg
Endzustand ∈ F	add_edge("name", END)	Markiert den Ausstieg
Übergang δ	add_edge("von", "nach")	Verbindet zwei Stationen

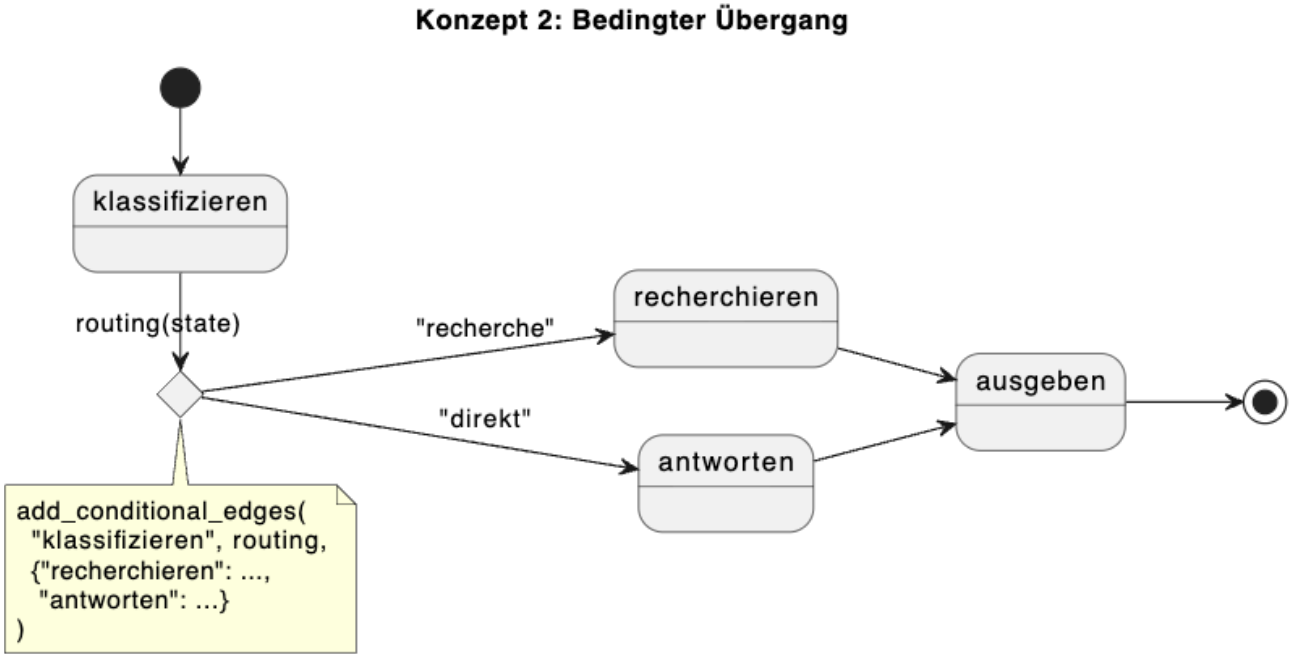
Merkregel
Jeder add_node-Aufruf ist ein Kreis im Diagramm. Jeder add_edge-Aufruf ist ein Pfeil.

Langgraph Beispiel 6: Graph als Kontrollfluss

```
from typing import TypedDict
from langgraph.graph import StateGraph, START, END

class State(TypedDict):
    nachricht: str

def empfangen(state: State) -> dict:
    return {"nachricht": state["nachricht"]}

def verarbeiten(state: State) -> dict:
    return {"nachricht": state["nachricht"].upper()}

def ausgeben(state: State) -> dict:
    return {"nachricht": f"Ergebnis: {state['nachricht']}"}

# --- Graph aufbauen ---
graph = StateGraph(State)
graph.add_node("empfangen", empfangen)       # Knoten
graph.add_node("verarbeiten", verarbeiten)
graph.add_node("ausgeben", ausgeben)

graph.add_edge(START, "empfangen")            # START → erster Knoten
graph.add_edge("empfangen", "verarbeiten")    # Kante
graph.add_edge("verarbeiten", "ausgeben")     # Kante
graph.add_edge("ausgeben", END)               # letzter Knoten → END
app = graph.compile()
```

7 Konzept 2: Bedingte Übergänge

Nicht jeder Weg ist fest. Manche Übergänge hängen von einer Bedingung ab — der Graph verzweigt sich. In der Automatentheorie wird der Übergang durch das Eingabesymbol bestimmt. In LangGraph übernimmt eine Routing-Funktion diese Rolle: Sie inspiziert den aktuellen Datenzustand und entscheidet, welcher Knoten als nächstes kommt.



Auffällig: Nach der Recherche geht es direkt zur Ausgabe — ohne zu prüfen, ob das Ergebnis ausreicht. Wie man den Graphen so erweitert, dass er bei Bedarf nochmal recherchiert, zeigt Konzept 4 (Zyklen).

Automatentheorie	LangGraph-Code	Bedeutung
Übergangsfunktion δ(q, a)	routing(state) → str	Entscheidet den nächsten Knoten
Eingabesymbol a	Rückgabewert der Routing-Fn.	Bestimmt den Pfad
Übergangstabelle	Mapping-Dictionary {...}	Ordnet Rückgabewerte Knoten zu

Merkregel
Jede Raute (Entscheidung) im Flussdiagramm wird zu einem **add_conditional_edges**-Aufruf.
Die Routing-Funktion beantwortet die Frage in der Raute.

Wichtiger Hinweis:
In den Automatendiagrammen aus Teil I steckt die Bedingung im Kantenlabel — jeder beschriftete Pfeil ist bereits ein bedingter Übergang. In der Software-Notation, die wir ab jetzt verwenden, wird der Entscheidungspunkt als Raute (◇) explizit gemacht.
Beides meint dasselbe: die Übergangsfunktion δ entscheidet, wohin es weitergeht.

In der klassischen Automatentheorie gibt es keine Rauten. Die Bedingung steckt **im Kantenlabel**: der Pfeil von "Zu" nach "Offen" ist beschriftet mit "bezahlen" — das *ist* die Bedingung. Jede Kante in einem Zustandsdiagramm ist implizit bedingt. Eine Raute wäre in dieser Notation redundant und formtheoretisch falsch.
Dass LangGraph-Code trotzdem **add_conditional_edges** braucht, ist ein **Implementationsdetail**: LangGraph weiß nicht von alleine, welche Kante es nehmen soll — es braucht eine Router-Funktion, die das zur Laufzeit entscheidet.
Der Automat in der Theorie hat diese Entscheidungslogik in der Übergangsfunktion δ eingebaut.

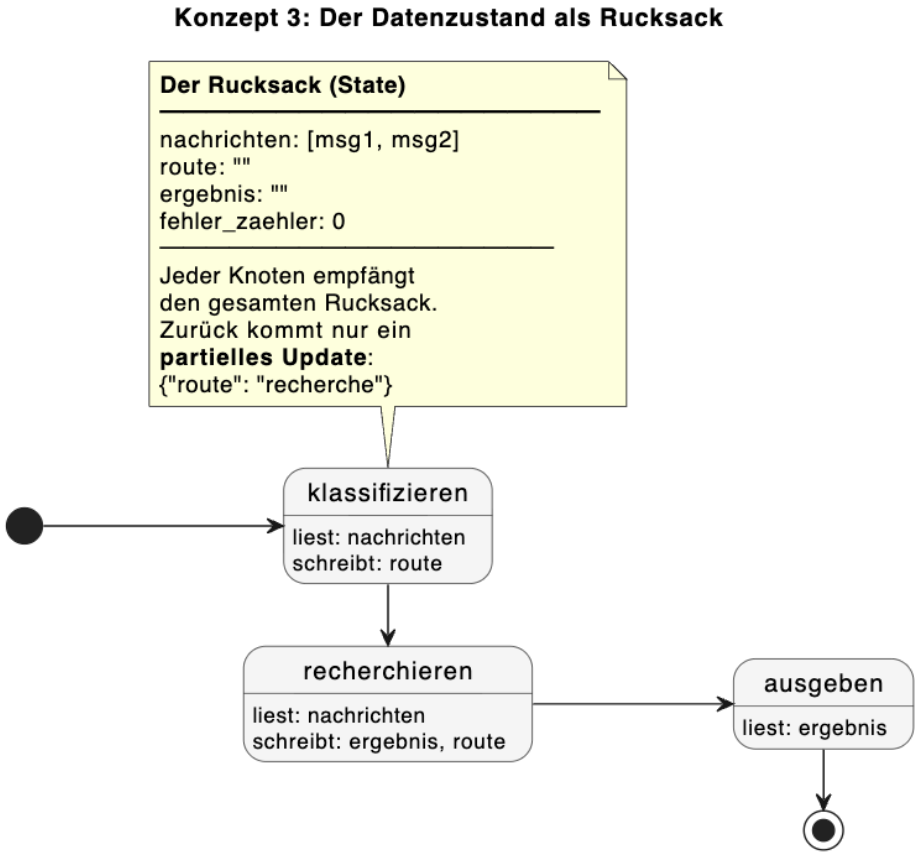
Der Bezug zu Mealy wird hier besonders deutlich: Die Routing-Funktion ist eine Funktion λ(q, D), die vom aktuellen Kontrollzustand (welcher Knoten) UND vom Datenzustand (was steht im State) abhängt — genau wie die Ausgabefunktion eines Mealy-Automaten.

Langgraph Beispiel 7: Der bedingte Übergang

```
def klassifizieren(state: State) -> dict:
    if "suche" in state["nachricht"].lower():
        return {"route": "recherche"}
    return {"route": "direkt"}

def routing(state: State) -> Literal["recherchieren", "antworten"]:
    """Die Routing-Funktion – die Übergangsfunktion δ."""
    if state["route"] == "recherche":
        return "recherchieren"
    return "antworten"

# --- Graph aufbauen ---
graph = StateGraph(State)
graph.add_node("klassifizieren", klassifizieren)
graph.add_node("recherchieren", recherchieren)
graph.add_node("antworten", antworten)
graph.add_node("ausgeben", ausgeben)

graph.add_edge(START, "klassifizieren")

# Bedingter Übergang
graph.add_conditional_edges(
    "klassifizieren",              # Ausgangsknoten
    routing,                       # Routing-Funktion
    {                              # Mapping
        "recherchieren": "recherchieren",
        "antworten": "antworten"
    }
)
graph.add_edge("recherchieren", "ausgeben")
graph.add_edge("antworten", "ausgeben")
graph.add_edge("ausgeben", END)
```

8 Konzept 3: Der Datenzustand als Rucksack

Durch den Graphen wandert ein Datenobjekt — der State. Jeder Knoten darf hineingreifen, lesen und verändern. Bedingte Übergänge schauen ebenfalls in diesen Rucksack. Das ist der Datenzustand des erweiterten Zustandsautomaten aus Kapitel 5.



Hinweis:

Wie sich der Rucksack bei Updates verhält — ob ein Feld überschrieben oder erweitert wird — hängt davon ab, ob ein Reducer definiert ist.

Dieses zentrale Konzept wird im folgenden Kapitel 8.1 ausführlich behandelt.

Merkregel
Der State ist der Rucksack. Knoten packen Dinge hinein und nehmen Dinge heraus. Routing-Funktionen schauen hinein, ohne etwas zu verändern.

Langgraph Beispiel 8: State-Schema: Der Bauplan des Rucksacks

```
class AgentState(TypedDict):
    nachrichten: Annotated[list, add_messages]    # wird angehängt
    route: str                                    # Routing-Entscheidung
    ergebnis: str                                 # Zwischenergebnis
    fehler_zaehler: int
```

Jeder Knoten nimmt den gesamten State und gibt ein partielles Update zurück (nur Felder, die sich ändern)

Langgraph Beispiel 9: Knotenfunktion: Lesen und Schreiben

```
def recherchieren(state: AgentState) -> dict:
    frage = state["nachrichten"][-1]       # Lesen
    ergebnis = suche_im_web(frage)         # Arbeit verrichten
    return {                               # Partielles Update
        "ergebnis": ergebnis,
        "route": "fertig"
    }
```

8.1 Konzept 3.5: Reducer — Wenn der Rucksack wächst, statt sich zu leeren

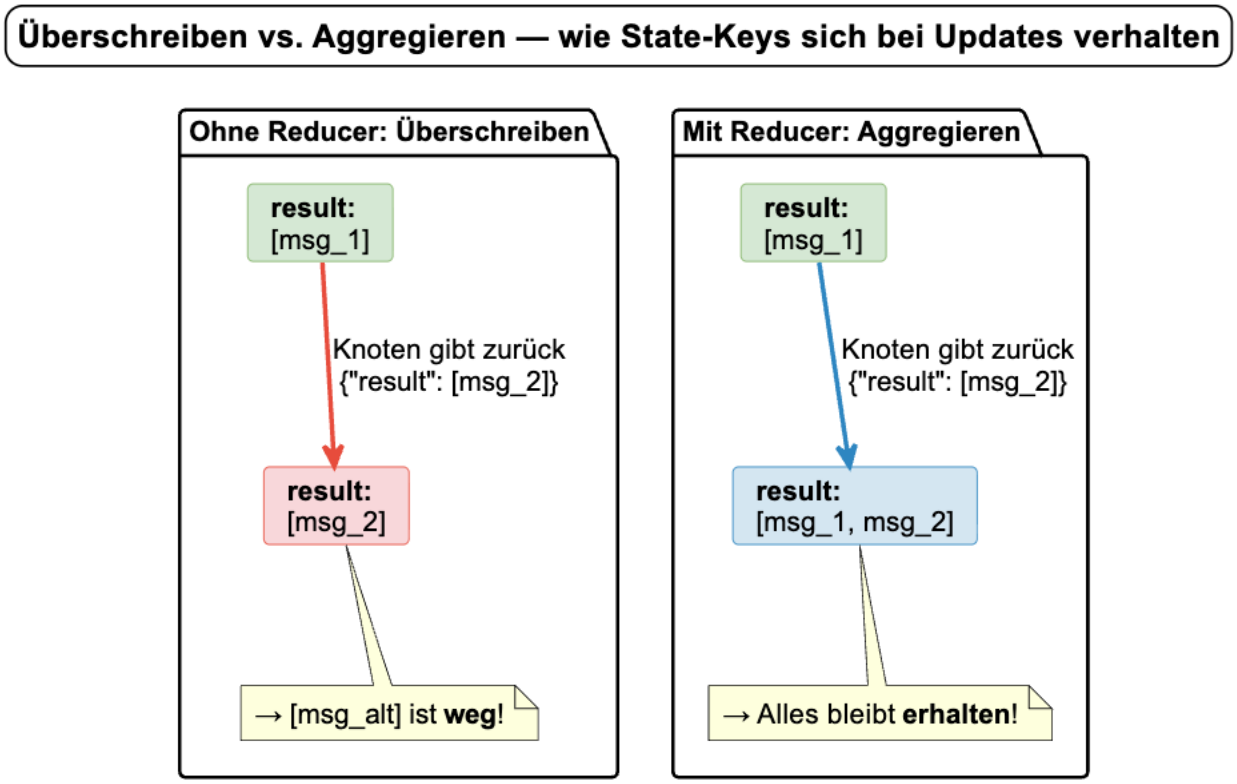
In den bisherigen Beispielen war das Verhalten des Rucksacks einfach: Ein Knoten gibt ein Feld zurück, und der neue Wert überschreibt den alten. Das ist das Standardverhalten in LangGraph — und für viele Felder genau richtig.

Aber was passiert mit dem Chatverlauf eines Agenten? Wenn ein Knoten eine neue Nachricht erzeugt und diese einfach den bisherigen Verlauf überschreibt, gehen alle vorherigen Nachrichten verloren. Das wäre, als würde man aus dem Rucksack alles herauswerfen, um ein einzelnes neues Ding hineinzulegen.

Reducer lösen genau dieses Problem. Ein Reducer ist eine Funktion, die festlegt, wie ein State-Feld aktualisiert wird. Statt den alten Wert blind zu ersetzen, kann ein Reducer den neuen Wert an den alten anhängen, zusammenführen oder auf andere Weise kombinieren.

Das Kernproblem: Überschreiben vs. Aggregieren

Das folgende Diagramm zeigt den Unterschied auf einen Blick:



Links (Überschreiben): Ein normaler State-Key verhält sich wie ein Briefkasten mit nur einem Platz — jeder neue Brief verdrängt den alten. Das Feld `result` enthält nach dem Update nur noch den neuen Wert.

Rechts (Aggregieren): Ein Reducer-Key verhält sich wie ein Tagebuch — jeder neue Eintrag wird angehängt, nichts geht verloren. Das Feld `result` wächst mit jedem Durchlauf.

Die Syntax: Das Annotated-Muster

In Python bindet das `Annotated`-Muster eine Reducer-Funktion an ein State-Feld. Die Reducer-Funktion bekommt den alten und den neuen Wert und entscheidet, was gespeichert wird:

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
import operator

class AgentState(TypedDict):
    # Ohne Reducer: neuer Wert ÜBERSCHREIBT den alten
    result: str
    route: str

    # Mit Reducer: neuer Wert wird ANGEHÄNGT
    result: Annotated[list, add_messages]
    outputs: Annotated[list[str], operator.add]
```

Gelesen wird das so: „Das Feld `result` ist eine Liste, und wenn ein Knoten neue Einträge zurückgibt, werden sie an die bestehende Liste angehängt (nicht ersetzt).“

Die gängigsten Reducer in LangGraph

Reducer	Verhalten	Typischer Einsatz
`add_messages`	Hängt neue Nachrichten an die bestehende Liste an	Chatverlauf eines Agenten
`operator.add`	Konkateniert Listen (Python-Listenaddition)	Ausgaben-Log, Zwischenergebnisse sammeln
`(kein Reducer)`	Neuer Wert überschreibt den alten	Routing-Entscheidung, aktuelles Ergebnis

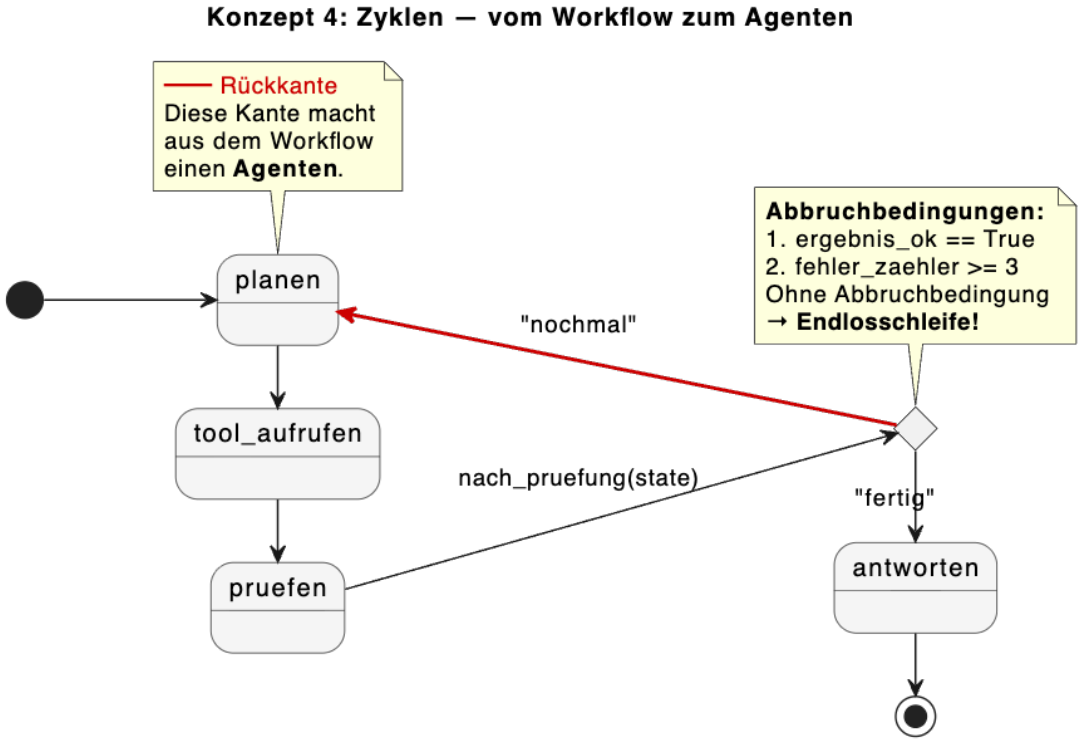
Wann brauche ich einen Reducer?
Die Faustregel ist einfach:

- Wenn ein Feld eine **wachsende Sammlung** darstellt (Nachrichten, Logs, Zwischenergebnisse), braucht es einen Reducer.
- Wenn ein Feld einen **aktuellen Zustandswert** darstellt (die aktuelle Route, den letzten Fehler, einen Zähler), ist Überschreiben das richtige Verhalten.

> *Merkregel*
> Ohne Reducer: Der Rucksack wird bei jedem Knoten geleert und neu befüllt. Mit Reducer: Der Rucksack wächst — nichts geht verloren. Die Frage beim State-Design lautet immer: „Soll dieser Wert die Geschichte erzählen (Reducer) oder nur den aktuellen Stand zeigen (kein Reducer)?“

9 Konzept 4: Zyklen — vom Workflow zum Agenten

Die bisherigen Graphen waren azyklisch — jeder Knoten wurde höchstens einmal besucht. Ein Agent entsteht erst durch Zyklen: versuchen, prüfen, bei Bedarf nochmal versuchen.
In der Automatentheorie ist ein Zyklus ein Pfad, der zu einem bereits besuchten Knoten zurückführt.



Langgraph Beispiel 10: Agent mit Retry-Zyklus

```
def pruefen(state: AgentState) -> dict:
    if state["ergebnis_ok"]:
        return {"route": "fertig"}
    if state["fehler_zaehler"] >= 3:
        return {"route": "fertig"}             # Abbruchbedingung!
    return {
        "route": "nochmal", "fehler_zaehler": state["fehler_zaehler"] + 1
    }

def nach_pruefung(state) -> Literal["planen", "antworten"]:
    if state["route"] == "nochmal":
        return "planen"                        # ← Zyklus: zurück zum Anfang
    return "antworten"                         # ← Ausstieg aus dem Zyklus

graph = StateGraph(AgentState)
graph.add_node("planen", planen)
graph.add_node("tool_aufrufen", tool_aufrufen)
graph.add_node("pruefen", pruefen)
graph.add_node("antworten", antworten)
graph.add_edge(START, "planen")
graph.add_edge("planen", "tool_aufrufen")
graph.add_edge("tool_aufrufen", "pruefen")
graph.add_conditional_edges(
    "pruefen", nach_pruefung, {"planen": "planen", "antworten": "antworten"})
graph.add_edge("antworten", END)
```

Langgraph Beispiel 11: Abbruchbedingungen

Jeder Zyklus braucht eine garantierte Ausstiegsbedingung. Ohne sie läuft der Agent endlos. Zwei Mechanismen stehen zur Verfügung:

```
# Mechanismus 1: Eigener Zähler im State
if state["fehler_zaehler"] >= 3:
    return {"route": "fertig"}

# Mechanismus 2: LangGraphs Rekursionslimit
app.invoke(state, {"recursion_limit": 10})
```

> *Merkregel*
> Ein Workflow ist ein azyklischer Graph. Ein Agent ist ein Graph mit mindestens einem Zyklus. Jeder Zyklus braucht eine Abbruchbedingung — sonst ist es kein Agent, sondern eine Endlosschleife.

10 Zusammenfassung: Die vollständige Mapping-Tabelle

Automatentheorie	Symbol	LangGraph	Code
Zustandsmenge Q	{ q_0, q_1, … qn }	Menge aller Knoten	add_node("name", fn)
Startzustand q_0	→	Einstiegspunkt	add_edge(START, "name")
Endzustände F	◉	Ausstiegspunkte	add_edge("name", END)
Eingabealphabet Σ	{ a, b, … }	Datenzustand	class State(TypedDict)
Fester Übergang	→	Feste Kante	add_edge("von", "nach")
Bedingter Übergang	→⟨B⟩	Bedingte Kante	add_conditional_edges(...)
Übergangsfunktion δ	δ: Q × Σ → Q	Routing-Funktion	def routing(state) -> str
Ausgabefunktion λ	λ: Q × Σ → Γ	Knotenfunktion	def knoten(state) -> dict
Erweiterter Zustand	(q, D)	Kontroll- + Datenzustand	Aktueller Knoten + State
Zyklus	Rückkante	Agent-Schleife	Kante zu früherem Knoten
Abbruchbedingung	—	Rekursionslimit	recursion_limit / Zähler

Automatentypus von LangGraph

LangGraph ist ein erweiterter Zustandsautomat mit Mealy-Charakter.

Die Ausgabe (State-Update) jedes Knotens hängt davon ab:

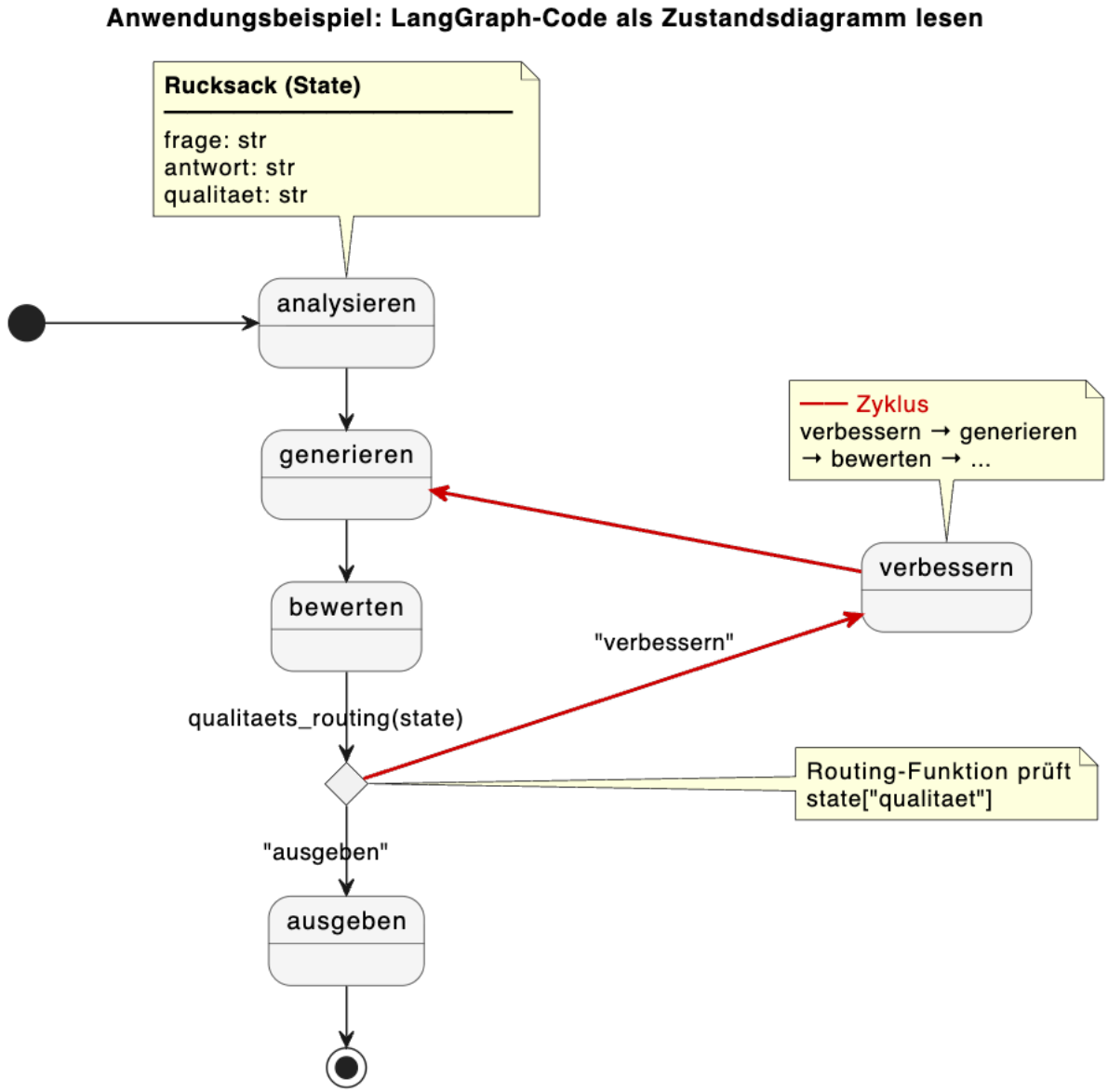
- welcher Knoten es ist
  UND
- was im Datenzustand steht.

Es ist weder ein reiner Moore- noch ein reiner Mealy-Automat, weil der Datenzustand kein endliches Eingabealphabet ist, sondern ein strukturiertes, potenziell unbegrenztes Datenobjekt.

11 Einen LangGraph-Code lesen — in vier Schritten

Jedes LangGraph-Programm lässt sich systematisch als Zustandsdiagramm lesen:

- **Schritt 1 — Knoten identifizieren:** Jeder add_node("name", funktion) ist ein Zustand. Zeichnen Sie einen Kreis und beschriften Sie ihn.
- **Schritt 2 — Kanten zeichnen:** Jeder add_edge ist ein Pfeil. Jeder add_conditional_edges ist eine Verzweigung — zeichnen Sie eine Raute mit beschrifteten Ausgängen.
- **Schritt 3 — State-Schema lesen:** Die TypedDict-Klasse verrät, was im Rucksack steckt. Notieren Sie die Felder neben dem Diagramm.
- **Schritt 4 — Zyklen und Abbruch finden:** Gibt es einen Pfeil, der zu einem früheren Knoten zurückführt? Wenn ja: Wo ist die Abbruchbedingung?



Langgraph Beispiel 12: Anwendungsbeispiel

```
class State(TypedDict):
    frage: str
    antwort: str
    qualitaet: str

graph = StateGraph(State)
graph.add_node("analysieren", analysieren)
graph.add_node("generieren", generieren)
graph.add_node("bewerten", bewerten)
graph.add_node("verbessern", verbessern)
graph.add_node("ausgeben", ausgeben)

graph.add_edge(START, "analysieren")
graph.add_edge("analysieren", "generieren")
graph.add_edge("generieren", "bewerten")
graph.add_conditional_edges(
    "bewerten", qualitaets_routing,
    {"verbessern": "verbessern", "ausgeben": "ausgeben"}
)
graph.add_edge("verbessern", "generieren")  # ← Zyklus!
graph.add_edge("ausgeben", END)
```

Teil III: Übungsaufgaben

Die Aufgaben sind nach Typ geordnet und innerhalb jedes Typs nach aufsteigender Komplexität. Jeder Typ trainiert eine andere kognitive Fähigkeit. Es wird empfohlen, die Aufgaben der Reihe nach zu bearbeiten.

Typ A: Code → Diagramm

Sie erhalten LangGraph-Code und zeichnen das zugehörige Zustandsdiagramm. Markieren Sie Start, Ende, alle Übergänge und den Datenzustand (Rucksack-Felder).

Aufgabe A1: Linearer Graph (Aufwärmung)
Zeichnen Sie das Zustandsdiagramm für den Code. Beschriften Sie alle Knoten und Kanten.

```
class State(TypedDict):
    text: str

graph = StateGraph(State)
graph.add_node("laden", laden)
graph.add_node("bereinigen", bereinigen)
graph.add_node("speichern", speichern)

graph.add_edge(START, "laden")
graph.add_edge("laden", "bereinigen")
graph.add_edge("bereinigen", "speichern")
graph.add_edge("speichern", END)
```

- *Hinweis: Drei Knoten, keine Verzweigung, kein Zyklus.*

Aufgabe A2: Einfache Verzweigung
Zeichnen Sie das Zustandsdiagramm. Markieren Sie bedingte Kanten und beschriften die Pfade.

```
class State(TypedDict):
    text: str
    sprache: str
    uebersetzung: str

graph = StateGraph(State)
graph.add_node("erkennen", sprache_erkennen)
graph.add_node("uebersetzen", uebersetzen)
graph.add_node("formatieren", formatieren)

graph.add_edge(START, "erkennen")
graph.add_conditional_edges(
    "erkennen", lambda s: "uebersetzen" if s["sprache"] != "de" else "formatieren",
    {"uebersetzen": "uebersetzen", "formatieren": "formatieren"}
)
graph.add_edge("uebersetzen", "formatieren")
graph.add_edge("formatieren", END)
```

- *Hinweis: Wo vereinigen sich die beiden Pfade wieder?*

Aufgabe A3: Verzweigung mit drei Pfaden

Zeichnen Sie das Zustandsdiagramm. Wie viele verschiedene Wege gibt es durch den Graphen?

```
class State(TypedDict):
    anfrage: str
    kategorie: str
    antwort: str

graph = StateGraph(State)
graph.add_node("eingang", eingang)
graph.add_node("technik", technik_handler)
graph.add_node("rechnung", rechnung_handler)
graph.add_node("allgemein", allgemein_handler)
graph.add_node("antwort", antwort_senden)

graph.add_edge(START, "eingang")
graph.add_conditional_edges(
    "eingang", kategorisieren,
     {"technik": "technik", "rechnung": "rechnung", "allgemein": "allgemein"}
)
graph.add_edge("technik", "antwort")
graph.add_edge("rechnung", "antwort")
graph.add_edge("allgemein", "antwort")
graph.add_edge("antwort", END)
```

▸ *Hinweis: Drei Pfade, die sich wieder vereinigen.*

Aufgabe A4: Graph mit Zyklus

Zeichnen Sie das Diagramm und identifizieren Sie:
(a) Zyklus (b) Abbruchbedingung (c) maximale Anzahl der Durchläufe

```
class State(TypedDict):
    aufgabe: str
    loesung: str
    versuche: int
    bestanden: bool

graph = StateGraph(State)
graph.add_node("loesen", loesen)
graph.add_node("testen", testen)
graph.add_node("korrigieren", korrigieren)
graph.add_node("abliefern", abliefern)

graph.add_edge(START, "loesen")
graph.add_edge("loesen", "testen")
graph.add_conditional_edges(
    "testen",
    lambda s: "abliefern" if s["bestanden"] or s["versuche"] >= 5
              else "korrigieren",
    {"abliefern": "abliefern", "korrigieren": "korrigieren"}
)
graph.add_edge("korrigieren", "loesen")
graph.add_edge("abliefern", END)
```

▸ *Hinweis: Die Abbruchbedingung hat zwei Teile — finden Sie beide.*

Aufgabe A5: Verschachtelte Entscheidungen

Zeichnen Sie das Zustandsdiagramm. Wie viele verschiedene Pfade gibt es von START bis END?

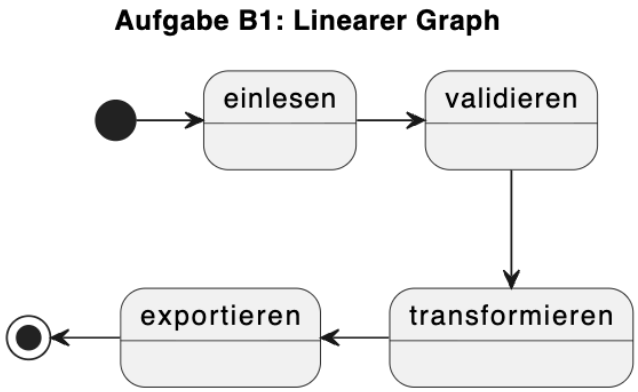
```
class State(TypedDict):
    query: str
    is_complex: bool
    needs_review: bool
    result: str

graph = StateGraph(State)
graph.add_node("analyze", analyze)
graph.add_node("simple_answer", simple_answer)
graph.add_node("deep_research", deep_research)
graph.add_node("review", review)
graph.add_node("deliver", deliver)

graph.add_edge(START, "analyze")
graph.add_conditional_edges(
    "analyze",
    lambda s: "deep_research" if s["is_complex"] else "simple_answer",
    {"deep_research": "deep_research", "simple_answer": "simple_answer"}
)
graph.add_edge("simple_answer", "deliver")
graph.add_conditional_edges(
    "deep_research",
    lambda s: "review" if s["needs_review"] else "deliver",
    {"review": "review", "deliver": "deliver"}
)
graph.add_edge("review", "deliver")
graph.add_edge("deliver", END)
```

▸ *Hinweis: Es gibt zwei Entscheidungspunkte — einer ist dem anderen nachgelagert.*

Typ B: Diagramm → Code

Sie erhalten ein Zustandsdiagramm und schreiben den zugehörigen LangGraph-Code. Definieren Sie das State-Schema und alle Kanten.

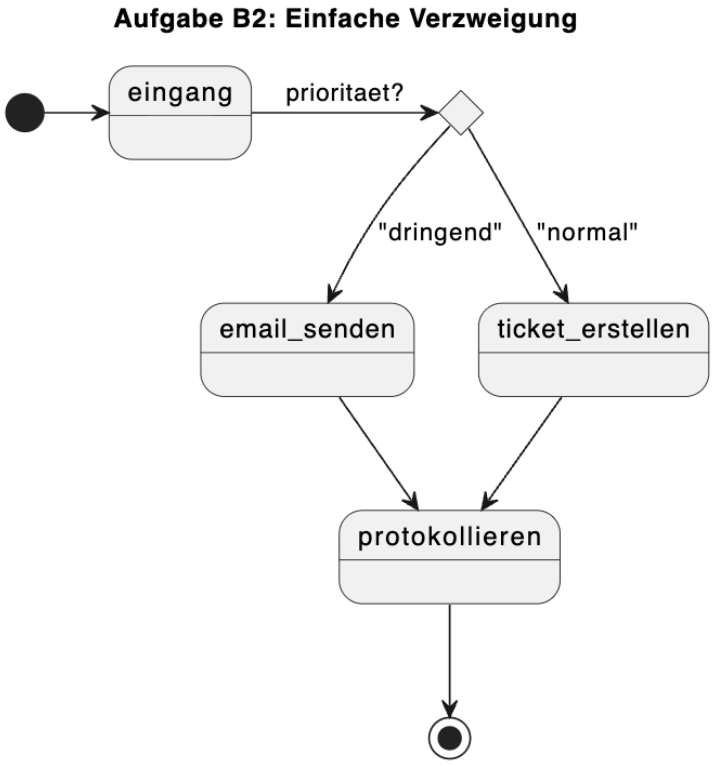
Aufgabe B1: Linearer Graph mit vier Stationen
Übersetzen Sie das folgende Diagramm in LangGraph-Code:



Definieren Sie ein passendes State-Schema.

- *Hinweis: Vier add_node-Aufrufe, vier add_edge-Aufrufe.*

Aufgabe B2: Einfache Verzweigung
Übersetzen Sie dieses Diagramm in LangGraph-Code:

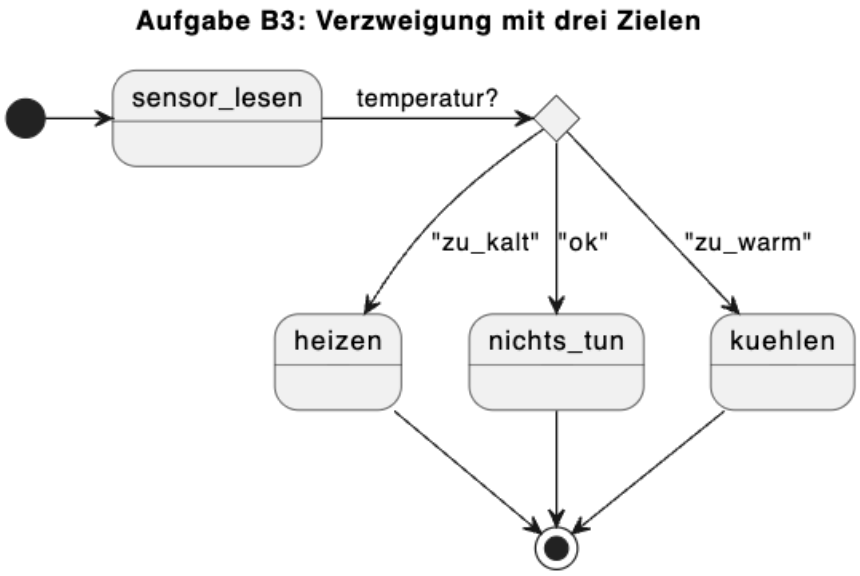


Die Routing-Funktion soll das Feld 'prioritaet' im State prüfen.

- *Hinweis: Beide Pfade führen zum selben Knoten 'protokollieren'.*

Aufgabe B3: Verzweigung mit drei Zielen

Übersetzen Sie dieses Diagramm in LangGraph-Code:

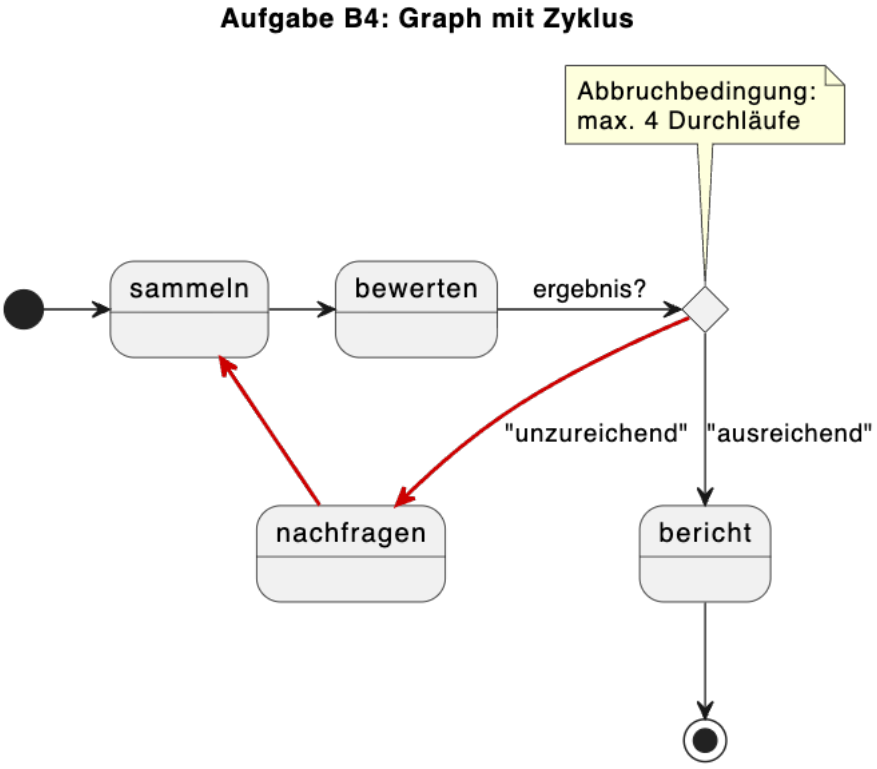


Definieren Sie die Routing-Funktion, die den Temperaturwert aus dem State prüft.

- *Hinweis: Der State braucht ein Feld für den Temperaturwert.*

Aufgabe B4: Graph mit Zyklus

Übersetzen Sie dieses Diagramm mit Zyklus in LangGraph-Code:

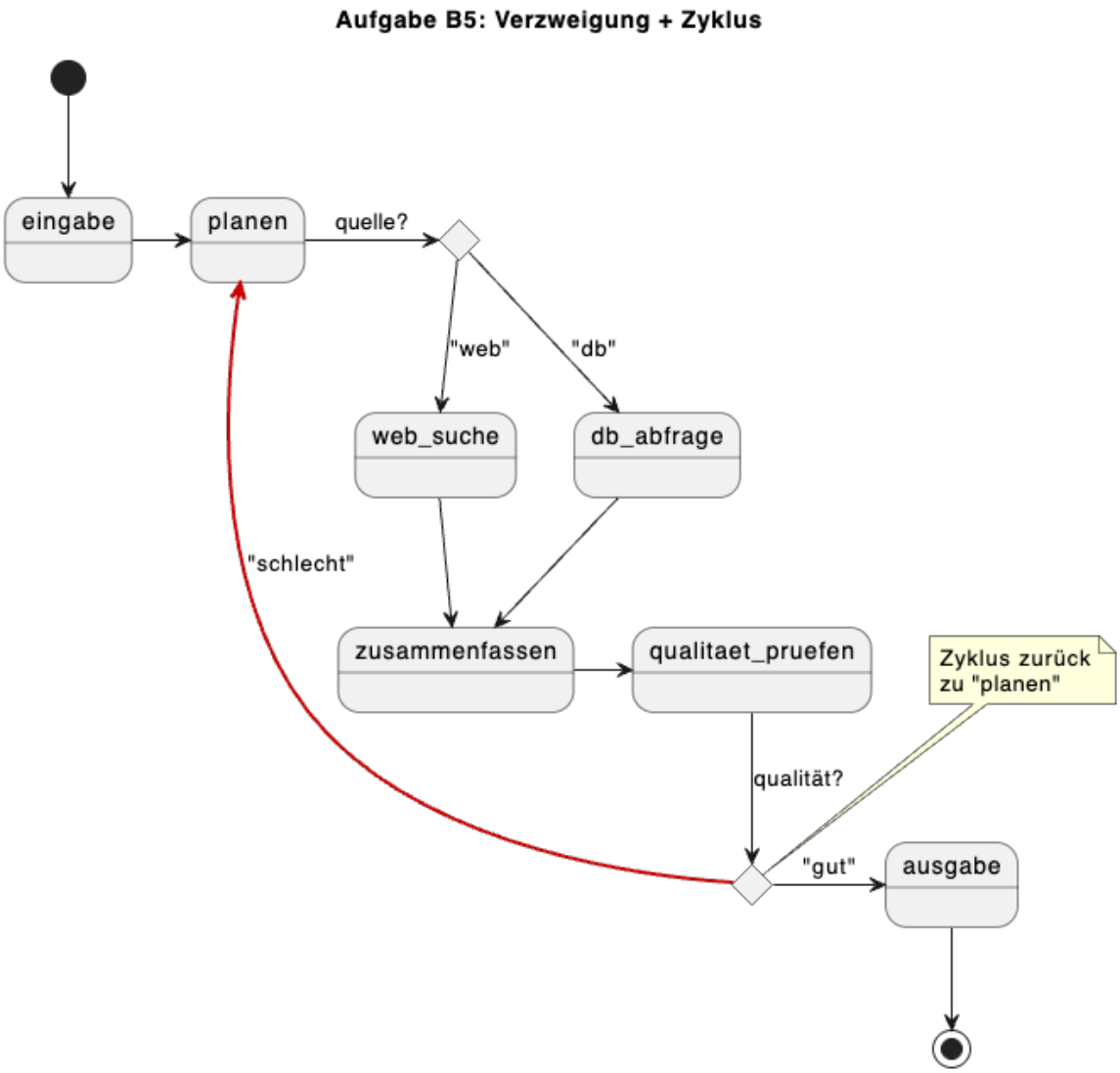


Fügen Sie eine Abbruchbedingung hinzu (max. 4 Durchläufe).

- *Hinweis: Sie brauchen ein Zählerfeld im State.*

Aufgabe B5: Komplexer Graph mit Zyklus und Verzweigung

Übersetzen Sie dieses Diagramm in LangGraph-Code:



Definieren Sie das vollständige State-Schema und alle Routing-Funktionen.

- *Hinweis: Dieser Graph hat sowohl eine Verzweigung als auch einen Zyklus.*

Typ C: Szenario → Graph → Code-Struktur

Sie erhalten eine Beschreibung in natürlicher Sprache und entwickeln daraus: (1) das Zustandsdiagramm, (2) das State-Schema, (3) die Graph-Struktur in LangGraph.

Aufgabe C1: E-Mail-Sortierer (einfach)
Ein System empfängt E-Mails und soll sie automatisch sortieren:
• Schritt 1: E-Mail einlesen und Absender extrahieren
• Schritt 2: Kategorie bestimmen (Privat oder Geschäftlich)
• Schritt 3a: Private Mails in den Ordner 'Persönlich' verschieben
• Schritt 3b: Geschäftliche Mails in den Ordner 'Arbeit' verschieben
• Schritt 4: Bestätigung protokollieren

Aufgaben:
(1) Zeichnen Sie das Zustandsdiagramm.
(2) Definieren Sie das State-Schema.
(3) Schreiben Sie die Graph-Struktur.

▸ Hinweis: Welche Felder braucht der Rucksack mindestens?

Aufgabe C2: Support-System (mittel)
Ein Kunde schreibt eine Support-Anfrage. Das System soll:

• Die Anfrage in eine Kategorie einordnen (Rechnung, Technik, Allgemein)
• Je nach Kategorie einen spezialisierten Agenten aufrufen
• Am Ende eine Zusammenfassung an den Kunden schicken

Aufgaben:
(1) Zeichnen Sie das Zustandsdiagramm.
(2) Definieren Sie das State-Schema.
(3) Identifizieren Sie die Routing-Funktion.
(4) Schreiben Sie die Graph-Struktur.

▸ *Hinweis: Drei spezialisierte Agenten = drei Pfade, die sich wieder vereinigen.*

Aufgabe C3: Recherche-Agent (komplex)
Ein Recherche-Agent soll:

• Eine Frage entgegennehmen
• Mehrere Quellen durchsuchen (Web, Datenbank, lokale Dokumente)
• Die Ergebnisse zusammenfassen
• Prüfen, ob die Antwort vollständig ist
• Falls nicht: gezielt nachrecherchieren (maximal 3 Durchläufe)
• Am Ende die finale Antwort ausgeben

Aufgaben:
(1) Zeichnen Sie das Zustandsdiagramm mit Zyklus.
(2) Definieren Sie das State-Schema.
(3) Wo ist die Abbruchbedingung?
(4) Schreiben Sie die Graph-Struktur.

▸ *Hinweis: Zyklus umfasst: zusammenfassen → prüfen → nachrecherchieren → zusammenfassen.*

Typ D: Fehlersuche

Der folgende Code enthält jeweils einen oder mehrere Fehler im Graphen (nicht in der Python-Syntax). Finden und erklären Sie den Fehler.

Aufgabe D1: Fehlende Terminierung

Was passiert, wenn dieser Graph kompiliert und ausgeführt wird?

```
graph = StateGraph(State)
graph.add_node("start_verarbeitung", starten)
graph.add_node("analyse", analysieren)
graph.add_node("synthese", synthetisieren)

graph.add_edge(START, "start_verarbeitung")
graph.add_edge("start_verarbeitung", "analyse")
graph.add_edge("analyse", "synthese")
# Fehlt etwas?
```

▸ *Hinweis: Zeichnen Sie den Graphen — hat er einen Endpunkt?*

Aufgabe D2: Unerreichbarer Knoten

Dieser Graph hat einen strukturellen Fehler. Welchen?

```
graph = StateGraph(State)
graph.add_node("eingang", eingang)
graph.add_node("pfad_a", pfad_a)
graph.add_node("pfad_b", pfad_b)
graph.add_node("pfad_c", pfad_c)
graph.add_node("ausgabe", ausgabe)

graph.add_edge(START, "eingang")
graph.add_conditional_edges(
    "eingang", routing, {"a": "pfad_a", "b": "pfad_b"}  # Wo ist pfad_c?
)
graph.add_edge("pfad_a", "ausgabe")
graph.add_edge("pfad_b", "ausgabe")
graph.add_edge("pfad_c", "ausgabe")
graph.add_edge("ausgabe", END)
```

▸ *Hinweis: Ein Knoten existiert, ist aber nie erreichbar.*

Aufgabe D3: Endlosschleife

Dieser Graph soll einen Retry-Mechanismus implementieren. Was ist das Problem?

```
def retry_routing(state) -> Literal["erneut", "fertig"]:
    if not state["erfolg"]:
        return "erneut"
    return "fertig"

graph = StateGraph(State)
graph.add_node("versuch", versuchen)
graph.add_node("ausgabe", ausgeben)

graph.add_edge(START, "versuch")
graph.add_conditional_edges(
    "versuch", retry_routing,
    {"erneut": "versuch", "fertig": "ausgabe"}
)
graph.add_edge("ausgabe", END)
```

▸ *Hinweis: Was passiert, wenn 'versuchen' niemals erfolg: True setzt?*

Aufgabe D4: Unvollständiges Routing

Finden Sie zwei Probleme in diesem Code:

```
class State(TypedDict):
    nachricht: str
    kategorie: str

def routing(state: State) -> str:
    return state["kategorie"]

graph = StateGraph(State)
graph.add_node("klassifizieren", klassifizieren)
graph.add_node("technik", technik_handler)
graph.add_node("rechnung", rechnung_handler)

graph.add_edge(START, "klassifizieren")
graph.add_conditional_edges(
    "klassifizieren", routing,
    {"technik": "technik", "rechnung": "rechnung"}
)
graph.add_edge("technik", END)
# Schauen Sie genau hin ...
```

▸ *Hinweis: Problem 1: Eine Kante fehlt. Problem 2: Was passiert bei unbekannter Kategorie?*

Aufgabe D5: Zyklus überspringt Verarbeitung
Dieser Agent soll bei jedem Retry-Durchlauf die Daten neu aufbereiten. Tut er das?

```
graph = StateGraph(State)
graph.add_node("daten_laden", daten_laden)
graph.add_node("aufbereiten", aufbereiten)
graph.add_node("analysieren", analysieren)
graph.add_node("pruefen", pruefen)
graph.add_node("ausgabe", ausgabe)

graph.add_edge(START, "daten_laden")
graph.add_edge("daten_laden", "aufbereiten")
graph.add_edge("aufbereiten", "analysieren")
graph.add_edge("analysieren", "pruefen")
graph.add_conditional_edges(
    "pruefen", check_quality,
    {"retry": "analysieren", "done": "ausgabe"}
)
graph.add_edge("ausgabe", END)
```

▸ *Hinweis: Zeichnen Sie den Zyklus: welche Knoten werden wiederholt, welche übersprungen?*

Aufgabe D6: State-Feld nie gesetzt
Dieser Code läuft, aber ein subtiler logischer Fehler steckt im State-Design. Welcher?

```
class State(TypedDict):
    frage: str
    braucht_recherche: bool
    ergebnis: str

def eingang(state: State) -> dict:
    return {"frage": state["frage"]}  # braucht_recherche wird nie gesetzt!

def routing(state: State) -> str:
    if state["braucht_recherche"]:  # KeyError oder False?
        return "recherche"
    return "direkt"

graph = StateGraph(State)
graph.add_node("eingang", eingang)
graph.add_node("recherche", recherche)
graph.add_node("direkt", direkt)

graph.add_edge(START, "eingang")
graph.add_conditional_edges(
    "eingang", routing,
    {"recherche": "recherche", "direkt": "direkt"}
)
graph.add_edge("recherche", END)
graph.add_edge("direkt", END)
```

▸ *Hinweis: Was ist der Defaultwert eines bool-Felds in einem TypedDict, das nie explizit gesetzt wird?*

Typ E: Erweiterung und Modifikation

Sie erhalten einen bestehenden Graphen und müssen ihn um neue Funktionalität erweitern. Zeichnen Sie das neue Diagramm und schreiben Sie die veränderte Graphstruktur.

Aufgabe E1: Qualitätsprüfung hinzufügen
Nehmen Sie den Graphen aus Aufgabe A2 (Spracherkennung / Übersetzung) und erweitern Sie ihn: Nach der Übersetzung soll eine Qualitätsprüfung stattfinden. Falls die Qualität unter einem Schwellwert liegt, soll erneut übersetzt werden (maximal 2 Wiederholungen).

Aufgaben:

(1) Zeichnen Sie das neue Diagramm.
(2) Welche neuen Felder braucht der Rucksack?
(3) Schreiben Sie die neue Graphstruktur.

- *Hinweis: Sie fügen einen Knoten und einen Zyklus hinzu.*

Aufgabe E2: Parallelisierung
Nehmen Sie den Support-Agenten aus C2 und ändern Sie die Architektur: Statt die Anfrage einem spezialisierten Agenten zuzuordnen, sollen ALLE drei Agenten parallel die Anfrage bearbeiten. Ein neuer Knoten 'beste_antwort_waehlen' wählt danach die beste Antwort aus.

Aufgaben:

(1) Zeichnen Sie das neue Diagramm.
(2) Wie ändert sich das State-Schema?
(3) Schreiben Sie die neue Graphstruktur.

- *Hinweis: Drei parallele Kanten, die sich in einem Knoten wieder vereinigen.*

Aufgabe E3: Eskalationspfad (Human-in-the-Loop)
Nehmen Sie den Support-Agenten aus C2 und erweitern Sie ihn um einen Eskalationspfad:
Wenn der spezialisierte Agent keine Lösung findet, soll die Anfrage an einen menschlichen Bearbeiter weitergeleitet werden.

Aufgaben:

(1) Wie verändert sich das Diagramm?
(2) Welche neuen Felder braucht der Rucksack?
(3) Was bedeutet 'Human-in-the-Loop' im Kontext eines Zustandsautomaten?

- *Hinweis: Ein neuer Knoten und eine bedingte Kante nach der Agentenbearbeitung.*

Aufgabe E4: Fehlerbehandlung nachrüsten
Nehmen Sie den linearen Graphen aus A1 (laden → bereinigen → speichern) und rüsten Sie eine Fehlerbehandlung nach:
• Jeder Knoten kann fehlschlagen (das Feld 'fehler' im State wird gesetzt)
• Bei Fehler soll ein Knoten 'fehler_behandeln' aufgerufen werden
• Nach der Fehlerbehandlung soll der fehlgeschlagene Knoten erneut versucht werden (max. 2x)

Aufgaben:

(1) Wie verändert sich das Diagramm drastisch?
(2) Wie viele Zyklen hat der neue Graph?
(3) Skizzieren Sie die Graphstruktur.

- *Hinweis: Jeder Knoten braucht eine eigene Fehlerkante.*

Exkurs: TypedDict — der Bauplan für den Rucksack

In allen bisherigen Beispielen definiert eine `TypedDict`-Klasse, welche Felder der State (der „Rucksack“) enthält. Dieses Kapitel erklärt, was ein TypedDict ist, warum LangGraph es verwendet und wie man es liest.

Was ist ein TypedDict?

Ein `TypedDict` ist eine Klasse aus dem Python-Modul `typing`. Sie beschreibt ein Dictionary mit festen Schlüsseln und festgelegten Typen — ähnlich wie ein Formular mit vorgedruckten Feldern. Anders als ein normales Dictionary (`dict`), bei dem jeder Schlüssel beliebig sein kann, legt ein TypedDict im Voraus fest: „Genau diese Felder gibt es, und sie haben genau diese Typen.“

	Normales dict	TypedDict
Schlüssel	beliebig, zur Laufzeit	fest definiert, zur Entwicklungszeit
Typen	nicht geprüft	annotiert und prüfbar
Neue Felder	jederzeit hinzufügbar	nur die definierten erlaubt
Analogie	leeres Blatt Papier	Formular mit Feldern

Warum nutzt LangGraph ein TypedDict?

LangGraph muss zu jedem Zeitpunkt wissen, welche Daten durch den Graphen fließen. Das TypedDict liefert dafür drei Garantien:

1.	**Struktur:** Jeder Knoten weiß, welche Felder er lesen und schreiben kann.
2.	**Dokumentation:** Wer den Code liest, sieht sofort den „Inhalt des Rucksacks“, ohne jede Knotenfunktion zu kennen.
3.	**Fehlervermeidung:** Typ-Checker wie mypy können Tippfehler und falsche Typen erkennen, bevor der Code läuft.

> Das TypedDict ist der Bauplan des Rucksacks. Es sagt nicht, was drin ist — das ändert sich bei jedem Knoten — sondern welche Fächer der Rucksack hat und was hineinpassen darf.

Beispiel 1: Support-Ticket

Ein einfacher Support-Agent verarbeitet Kundenanfragen. Der Rucksack braucht genau vier Fächer:

```
from typing import TypedDict

class SupportState(TypedDict):
    # Die Kundenanfrage als Text
    anfrage: str
    # Kategorie: 'technik', 'rechnung' oder 'allgemein'
    kategorie: str
    # Die generierte Antwort
    antwort: str
    # Wurde das Ticket gelöst?
    geloest: bool
```

Jedes Feld hat einen Namen und einen Typ. Die Knotenfunktionen greifen auf genau diese Felder zu:

```
def klassifizieren(state: SupportState) -> dict:
    # Liest 'anfrage', schreibt 'kategorie'
    if "rechnung" in state["anfrage"].lower():
        return {"kategorie": "rechnung"}
    return {"kategorie": "allgemein"}
```

Der Knoten `klassifizieren` liest das Feld `anfrage` und schreibt das Feld `kategorie`. Alle anderen Felder bleiben unverändert — das ist das Prinzip des partiellen Updates.

Beispiel 2: Recherche-Agent mit Verlauf

Ein Recherche-Agent durchsucht mehrere Quellen und sammelt Zwischenergebnisse. Hier zeigt sich, wie TypedDict und Reducer zusammenspielen:

```
from typing import TypedDict, Annotated
from langgraph.graph.message import add_messages
import operator

class RecherState(TypedDict):
    # ---- Felder OHNE Reducer (Überschreiben) ----
    frage: str
    route: str
    fehler_zaehler: int

    # ---- Felder MIT Reducer (Aggregieren) ----
    nachrichten: Annotated[list, add_messages]
    quellen: Annotated[list[str], operator.add]
```

Dieses TypedDict definiert fünf Fächer im Rucksack. Drei davon (`frage`, `route`, `fehler_zaehler`) werden bei jedem Update überschrieben. Zwei davon (`nachrichten`, `quellen`) wachsen dank Reducer mit jedem Durchlauf.

Feld	Typ	Reducer	Bedeutung
`frage`	`str`	—	Die aktuelle Nutzerfrage
`route`	`str`	—	Routing-Entscheidung
`fehler_zaehler`	`int`	—	Zähler für Retry-Logik
`nachrichten`	`list`	`add_messages`	Wachsender Chatverlauf
`quellen`	`list[str]`	`operator.add`	Gesammelte Quellen-URLs

Ein TypedDict lesen — in drei Schritten

Wenn Sie in einem LangGraph-Programm auf ein TypedDict stoßen, gehen Sie so vor:

Schritt 1	**Felder auflisten:** Welche Schlüssel gibt es? Das sind die „Fächer“ des Rucksacks.
Schritt 2	**Typen prüfen:** Welchen Typ hat jedes Feld? Gibt es `Annotated`-Felder? Wenn ja: dort steckt ein Reducer (siehe Konzept 3.5).
Schritt 3	**Datenfluss nachvollziehen:** Welcher Knoten liest welches Feld? Welcher Knoten schreibt es? Notieren Sie den Fluss neben dem Zustandsdiagramm.

Visualisierung: Datenfluss durch den Rucksack

Die folgende Tabelle zeigt für jeden Knoten des Recherche-Agenten, welche Felder des TypedDict er liest und welche er schreibt. Felder mit Reducer (⊕) sind blau hervorgehoben — dort werden Werte angehängt, nicht überschrieben.

Knoten ↓ / Feld →	`frage`	`route`	`fehler_zaehler`	`nachrichten` ⊕ Reducer	`quellen` ⊕ Reducer
`empfangen`	← liest	→ schreibt	—	—	—
`recherchieren`	← liest	—	—	→ schreibt ⊕	→ schreibt ⊕
`prüfen`	—	↔ liest & schreibt	↔ liest & schreibt	—	—
`antworten`	—	—	—	→ schreibt ⊕	—

Legende:

← liest	Der Knoten liest dieses Feld aus dem State
→ schreibt	Der Knoten überschreibt dieses Feld (kein Reducer)
→ schreibt ⊕	Der Knoten hängt einen Wert an (Reducer-Feld)
↔ liest & schreibt	Der Knoten liest und überschreibt das Feld
—	Der Knoten greift nicht auf dieses Feld zu

So liest man die Tabelle: Jede Zeile ist ein Knoten, jede Spalte ein Feld im Rucksack. Die Zelle zeigt, ob der Knoten das Feld liest, schreibt oder beides. Leere Zellen (—) bedeuten: kein Zugriff. Zum Beispiel liest der Knoten `recherchieren` das Feld `frage` und hängt Ergebnisse an `nachrichten` und `quellen` an (⊕), ohne die bisherigen Einträge zu löschen.

> *Merkregel*
> Wer den Datenfluss eines LangGraph-Agenten verstehen will, braucht zwei Dinge: das Zustandsdiagramm (welcher Knoten folgt auf welchen?) und die Datenfluss-Tabelle (welcher Knoten greift auf welches Feld zu?). Beides zusammen ergibt das vollständige Bild.

Häufige Fehler beim State-Design

Fehler	Folge
Feld im TypedDict vergessen	KeyError zur Laufzeit, wenn ein Knoten darauf zugreift
Feld nie beschrieben	Routing-Funktionen erhalten None oder den Initialwert — subtile Logikfehler (vgl. Aufgabe D6)
Reducer vergessen bei Listen	Nur die letzte Nachricht überlebt — der gesamte Verlauf geht verloren
Falschen Typ annotiert	Kein Laufzeitfehler (Python prüft Typen nicht), aber Typ-Checker melden Warnungen

Glossar

Begriff	Definition	LangGraph-Entsprechung
Zustand (State)	Diskrete Station im Ablauf	*Node (Knoten)*
Übergang (Transition)	Gerichtete Verbindung zwischen Zuständen	*Edge (Kante)*
Startzustand (q_0)	Einstiegspunkt des Automaten	*START*
Endzustand ($\in F$)	Ausstiegspunkt des Automaten	*END*
Übergangsfunktion (δ)	Regel für den nächsten Zustand	*Routing-Funktion*
Ausgabefunktion (λ)	Erzeugt Ausgabe bei Übergang/Zustand	*Knotenfunktion*
Datenzustand	Strukturiertes Datenobjekt im Graphen	*State(TypedDict)*
Reducer	Aggregiert Zustandsupdates	*Annotated[list, add_messages]*
Zyklus	Pfad zurück zu früherem Knoten	*Rückkante*
Abbruchbedingung	Garantie der Terminierung	*Zähler / recursion_limit*
Moore-Automat	Ausgabe hängt nur vom Zustand ab	*— (konzeptuelle Grundlage)*
Mealy-Automat	Ausgabe hängt von Zustand + Eingabe ab	*— (konzeptuelle Grundlage)*
Erweiterter Automat	EFSM: Kontroll- + Datenzustand	*LangGraph StateGraph*