* [Chat](https://grok.com/)⌘J
* Voice
* [Imagine](https://grok.com/imagine)
* [Files](https://grok.com/files)

[Tasks](https://grok.com/tasks)

[Projects](https://grok.com/project)

 History

Yesterday

August

Hoe zet ik een ai agent framework op die een multi agent ondersteunt, rl orchestrating heeft, conext en geheugen opbouwt en werkt met verschillende onderliggende llm’s?

Thought for 59s

Om een AI-agentframework op te zetten dat ondersteuning biedt voor multi-agents, orchestratie met reinforcement learning (RL), opbouw van context en geheugen, en compatibiliteit met verschillende large language models (LLM's), zijn er diverse open-source opties beschikbaar op basis van recente ontwikkelingen in het veld. Ik zal me richten op de meest relevante frameworks, afgeleid uit betrouwbare bronnen, en een stapsgewijze handleiding geven voor de implementatie. De keuze hangt af van uw specifieke behoeften, maar Agent Lightning springt eruit vanwege de expliciete RL-integratie, terwijl Agno en LangGraph sterke ondersteuning bieden voor multi-agents, geheugen en meerdere LLM's. CrewAI en AutoGen zijn alternatieven voor eenvoudige multi-agent-orchestratie, hoewel RL-ondersteuning beperkter is.

**Aanbevolen Frameworks**

* **Agent Lightning**: Dit framework is ideaal voor RL-gebaseerde orchestratie van AI-agents, met ondersteuning voor multi-agents, context- en geheugenbeheer via een Markov Decision Process (MDP)-structuur, en compatibiliteit met meerdere LLM's (bijv. via API's zoals OpenAI of Google).

 Het maakt decoupling mogelijk tussen agent-uitvoering en RL-training, wat het flexibel maakt voor complexe workflows.

 **Agno**: Een full-stack framework voor multi-agent-systemen met ingebouwde geheugen- en contextopbouw, gedeelde kennis tussen agents, en ondersteuning voor meer dan 23 LLM-providers (bijv. Anthropic Claude en OpenAI GPT-modellen).

 RL-orchestratie wordt niet expliciet vermeld, maar het kan worden uitgebreid.

 **LangGraph (onderdeel van LangChain)**: Ondersteunt multi-agent-workflows met stateful geheugen voor langetermijncontext, orchestratie van agents, en integratie met meerdere LLM's. RL kan potentieel worden geïntegreerd via extensies, maar is niet native.**CrewAI**: Gericht op multi-agent-collaboratie en orchestratie, met LLM-compatibiliteit en contextbeheer, maar zonder expliciete RL.**AutoGen**: Een Microsoft-framework voor multi-agent-conversaties en orchestratie, met geheugenbeheer en ondersteuning voor verschillende LLM's. RL is niet primair, maar kan worden toegevoegd.

Voor een complete setup raad ik Agent Lightning aan als primaire keuze vanwege de RL-focus, met mogelijke integratie van Agno of LangGraph voor geavanceerde multi-agent-functies.

**Stapsgewijze Handleiding voor Opzetten (Voorbeeld met Agent Lightning)**

Deze stappen zijn gebaseerd op de framework-documentatie en veronderstellen een Python-omgeving. Zorg voor een recente Python-versie (bijv. 3.10+) en een virtuele omgeving om afhankelijkheden te beheren.

1. **Installatie van Vereisten**:
   * Installeer de benodigde bibliotheken via pip. Agent Lightning vereist RL-bibliotheken zoals Gym of RLlib, en LLM-interfaces.

text

* 
* pip install agent-lightning # Of clone van GitHub als niet publiek beschikbaar
* pip install torch gymnasium # Voor RL-ondersteuning

pip install openai anthropic # Voor LLM-integraties (pas aan op basis van gekozen modellen)

* Stel API-sleutels in voor LLM's (bijv. export OPENAI\_API\_KEY=uw-sleutel).

 **Framework Configureren**:

* Richt een Lightning Server in voor RL-training en een Client voor agent-uitvoering. Dit scheidt uitvoering van optimalisatie.
* Definieer agents als MDP's: States vertegenwoordigen context (bijv. huidige input en geschiedenis), actions zijn LLM-outputs, en rewards sturen RL-orchestratie.
* Voor multi-agents: Configureer meerdere agents (single-LLM met rollen of multi-LLM) en optimaliseer selectief via MARL (multi-agent RL).

 **Geheugen en Context Opbouwen**:

* Gebruik de unified data interface voor trajecten: Elke transitie bevat state (context), action, en reward. Dit bouwt automatisch geheugen op door sequenties te breken in beheersbare delen.
* Integreer observability-tools zoals OpenTelemetry voor tracing en langetermijngeheugen.

 **Multi-Agent en RL-Orchestratie Implementeren**:

* Creëer een agent-functie die interageert met een omgeving (bijv. een taak of simulatie).
* Voorbeeldcode voor een basis multi-agent setup met RL-training (aangepast uit documentatie):

python

* + import os
  + from agent\_lightning import Client, Resource, Task
  + from uw\_agent import agent\_function *# Vervang door uw agent-logica*
  + from environments import UwOmgeving *# Bijv. een custom RL-omgeving*
  + *# Server-client setup*
  + client = Client(os.environ["AgentLightningServerUrl"]) *# Stel server-URL in*
  + *# Upload dataset voor training (bijv. taken met ground truth)*
  + client.upload\_data("data/train.jsonl") *# JSONL-formaat met taken*
  + *# Definieer agent-run met multi-agent logica*
  + def agent\_run(resource: Resource, task: Task):
  + *# Multi-agent: Initialiseer agents met verschillende LLM's*
  + agent1 = agent\_function(resource.model\_api, "Rol1", model="gpt-4o") *# LLM1*
  + agent2 = agent\_function(resource.model\_api, "Rol2", model="claude-3") *# LLM2*
  + *# Orchestratie: Gebruik RL om interacties te coördineren*
  + state = {"context": task.input, "geheugen": []} *# Bouw context op*
  + for step in range(max\_steps):
  + action1 = agent1.act(state)
  + action2 = agent2.act(state)
  + reward = UwOmgeving.score(action1, action2, task.ground\_truth) *# RL-reward*
  + state["geheugen"].append((action1, action2, reward)) *# Geheugen update*
  + return reward *# Voor RL-optimalisatie*
  + *# Start training met multi-workers voor schaling*
  + client.train(agent\_run, nworkers=4) *# Optimaliseer met RL-algoritme (bijv. PPO)*
  + Dit voorbeeld ondersteunt RL-orchestratie (via LightningRL-algoritme), bouwt context op in de state, en gebruikt meerdere LLM's.

1. **Testen en Uitvoeren**:
   * Voer de client uit om agents te draaien en data te verzamelen voor RL-updates.
   * Monitor rewards en pas hyperparameters aan (bijv. learning rate in RL).
   * Voor schaling: Gebruik batch-accumulatie voor efficiënte contextbeheer in lange interacties.
2. **Integratie met Andere Frameworks**:
   * Combineer met Agno voor geavanceerd geheugen: Voeg Agno's Memory-drivers toe aan de state.
   * Voor LangGraph: Bouw workflows met langgraph voor stateful multi-agents, en integreer RL via custom nodes.

**Overwegingen en Beperkingen**

* **Compatibiliteit**: Alle genoemde frameworks werken met LLM's via API's, maar test op vendor lock-in.
* **RL-Specifiek**: Als RL cruciaal is, prioriteer Agent Lightning; anders volstaat LangGraph voor algemene orchestratie.
* **Resources**: Begin met eenvoudige taken om te valideren, en schaal op met cloud-resources (bijv. AWS voor Agent Squad-integraties).
* Raadpleeg de officiële documentatie voor updates, aangezien het veld snel evolueert. Als u specifieke code-aanpassingen nodig heeft, voorzie meer details over uw omgeving.

how to do this Framework Configureren:

* Richt een Lightning Server in voor RL-training en een Client voor agent-uitvoering. Dit scheidt uitvoering van optimalisatie.
* Definieer agents als MDP's: States vertegenwoordigen context (bijv. huidige input en geschiedenis), actions zijn LLM-outputs, en rewards sturen RL-orchestratie.
* Voor multi-agents: Configureer meerdere agents (single-LLM met rollen of multi-LLM) en optimaliseer selectief via MARL (multi-agent RL).

Thought for 59s

**Framework Configureren: Gedetailleerde Handleiding voor Agent Lightning**

Hieronder leg ik stap voor stap uit hoe je het framework configureert op basis van de officiële documentatie en het onderzoekspaper van Agent Lightning. Dit framework, ontwikkeld door Microsoft Research, scheidt de RL-training (via de Lightning Server) van de agent-uitvoering (via de Lightning Client) voor efficiënte optimalisatie. Agents worden gemodelleerd als Markov Decision Processes (MDP's), waarbij states context en geschiedenis vertegenwoordigen, actions LLM-outputs zijn, en rewards de orchestratie sturen. Voor multi-agents ondersteunt het selectieve optimalisatie en Multi-Agent Reinforcement Learning (MARL) voor coördinatie tussen agents.

Ik baseer dit op de architectuur uit het paper en praktische setup-instructies van de GitHub-repo.

**1. Richt een Lightning Server in voor RL-training en een Client voor agent-uitvoering**

Dit scheidt de uitvoering (client) van de optimalisatie (server), wat schaling en decoupling mogelijk maakt. De server beheert de RL-training, data en model-updates, terwijl de client agents draait en trajecten verzamelt.

**Stapsgewijze Setup:**

* **Vereisten Installeren:**
  + Maak een virtuele omgeving met Python 3.10+ (bijv. via conda create -n agent-lightning python=3.10 en conda activate agent-lightning).
  + Installeer core dependencies voor RL:

text

 pip install torch==2.7.0 torchvision==0.22.0 torchaudio==2.7.0 --index-url https://download.pytorch.org/whl/cu128

pip install flash-attn --no-build-isolation

pip install vllm==0.9.2

pip install verl==0.5.0

 Installeer Agent Lightning zelf: pip install agentlightning.

 Optioneel: Voeg agent-frameworks toe zoals AutoGen of LangGraph voor compatibiliteit:

text

* 
* pip install "autogen-agentchat" "autogen-ext[openai]"

pip install langgraph "langchain[openai]" langchain-community

 **Lightning Server Configureren en Starten:**

* De server fungeert als controller: hij beheert datasets, batcht taken, en exposeert een OpenAI-achtige API voor model-interacties.
* Configureer via een config-file (bijv. YAML of Python-script) met parameters zoals learning rate, batch size, en RL-algoritme (bijv. PPO of GRPO via LightningRL).
* Start de server in een terminal: Gebruik een script zoals lightning\_server.py (aangepast uit voorbeelden), bijv.:

text

* 
* from agentlightning import LightningServer
* server = LightningServer(
* model\_path="gpt-4o", # Of je LLM-model
* rl\_algorithm="LightningRL", # Hierarchische RL
* dataset\_path="data/train.jsonl" # Je trainingsdata
* )

server.start() # Luistert op een port, bijv. localhost:8000

* Zorg voor environment variables zoals API-keys voor LLM's en Ray (voor distributed computing): export RAY\_ADDRESS=auto.

 **Lightning Client Configureren en Starten:**

* De client handelt agent-uitvoering af, verzamelt data (trajecten) en stuurt ze terug naar de server.
* Start in een aparte terminal of achtergrondproces:

text

* + from agentlightning import LightningClient
  + client = LightningClient(
  + server\_url="http://localhost:8000", # Server-adres
  + agent\_function=je\_agent\_functie # Zie onder voor definitie
  + )
  + client.run() # Verzamelt data en stuurt trajecten
  + Voor schaling: Gebruik meerdere clients voor data-parallelisme (intra-node of inter-node).
* **Tips voor Scheiding en Optimalisatie:**
  + Lanceer server en client apart om decoupling te behouden.
  + Monitor timeouts: Stel limieten in op prompts/responses om hangs te voorkomen.
  + Sla checkpoints op voor herstel bij OOM-errors in VERL.

Dit setup zorgt ervoor dat optimalisatie (RL-training op server) onafhankelijk is van uitvoering (agents op client), met naadloze communicatie via API.

**2. Definieer agents als MDP's: States vertegenwoordigen context, actions zijn LLM-outputs, en rewards sturen RL-orchestratie**

Agents worden geformuleerd als MDP's voor een uniforme data-interface, wat RL-training vereenvoudigt. Elke agent-uitvoering is een sequentie van transities: state → action → reward.

**Hoe te Definiëren:**

* **States**: Dit is de huidige context, inclusief input, geschiedenis en semantische variabelen (bijv. user query, vorige outputs). Bijv. in een RAG-agent: state = {"user\_input": "...", "retrieved\_docs": [...], "history": [...] }.
* **Actions**: Outputs van de LLM, behandeld als discrete acties (bijv. gegenereerde tekst of tool-call).
* **Rewards**: Meet kwaliteit, bijv. gebaseerd op task-succes (F1-score, accuracy). Rewards kunnen intermediair of terminaal zijn, en sturen de orchestratie door voordelen te berekenen voor optimalisatie.

**Voorbeeldcode voor MDP-Definitie (aangepast uit paper-voorbeelden):**

python

from agentlightning import MDPTransition, Agent

def je\_agent\_functie(state):

*# State: Bouw context op met input en geschiedenis*

prompt = f"Context: {state['history']}\nHuidige input: {state['input']}"

*# Action: Roep LLM aan voor output*

action = llm.generate(prompt) *# Bijv. via OpenAI API*

*# Reward: Bereken op basis van outcome (bijv. custom scorer)*

reward = scorer.evaluate(action, state['ground\_truth'])

*# Retourneer transitie voor RL*

return MDPTransition(state=state, action=action, reward=reward)

*# Integreer in client*

client = LightningClient(..., agent\_function=je\_agent\_functie)

* Gebruik de unified data interface: Trajecten zijn lijsten van transities, geabstraheerd voor elk agent-type.
* Voor orchestratie: LightningRL (hierarchische RL) deelt rewards over transities voor token-level updates, zonder masking.

**3. Voor multi-agents: Configureer meerdere agents (single-LLM met rollen of multi-LLM) en optimaliseer selectief via MARL**

Agent Lightning ondersteunt multi-agent-systemen door selectieve optimalisatie van agents, met MARL voor inter-afhankelijkheden.

**Configuratie:**

* **Single-LLM met Rollen**: Eén LLM speelt meerdere rollen via prompts. Bijv. in text-to-SQL: Agents voor SQL-schrijven, checken en herschrijven.
  + Configureer: Definieer elke rol als sub-MDP in de state.
* **Multi-LLM**: Elke LLM als aparte agent, met MARL voor coördinatie (cooperatief of competitief).
  + Optimaliseer selectief: Kies welke agents te trainen (bijv. alleen schrijver en herschrijver).

**Voorbeeldcode voor Multi-Agent Setup:**

python

from agentlightning import MultiAgentClient, MARLOptimizer

*# Definieer agents*

agent1 = Agent(role="SQL Writer", model="gpt-4o") *# MDP voor schrijven*

agent2 = Agent(role="SQL Rewriter", model="claude-3") *# MDP voor herschrijven*

*# Multi-agent functie*

def multi\_agent\_run(state):

action1 = agent1.act(state) *# Action van agent1*

updated\_state = {"context": state['context'] + action1} *# Bouw gedeelde context*

action2 = agent2.act(updated\_state) *# Action van agent2*

reward = scorer.multi\_evaluate([action1, action2], state['ground\_truth']) *# Gedeelde reward*

return [MDPTransition(state, action1, reward), MDPTransition(updated\_state, action2, reward)]

*# Configureer met MARL*

client = MultiAgentClient(

server\_url="...",

agent\_function=multi\_agent\_run,

optimizer=MARLOptimizer(algorithm="LightningRL") *# Ondersteunt MARL voor inter-dependencies*

)

* MARL-Integratie: Modelleer elke agent als onafhankelijke MDP of gebruik MARL voor gedeelde rewards/advantages. Voorbeelden zoals Spider (multi-agent SQL) tonen verbeteringen in rewards.
* Schaling: Gebruik meerdere clients voor parallelle MARL-training.

**Overwegingen:** Begin met eenvoudige taken (bijv. Calc-X of Spider datasets) om te testen. Raadpleeg de GitHub-voorbeelden voor aanpassingen.

Als je specifieke code of datasets nodig hebt, geef meer details!

kun je dit alle,aal inclusief code in een md document zetten

Thought for 11s

**AI Agent Framework Opzetten: Multi-Agent met RL-Orchestratie, Context en Geheugen**

Dit document vat de volledige instructies samen voor het opzetten van een AI-agent framework dat multi-agents ondersteunt, reinforcement learning (RL) voor orchestratie gebruikt, context en geheugen opbouwt, en compatibel is met verschillende large lan

Prioriteer Agent Lightning voor RL-focus.

**Stapsgewijze Handleiding voor Opzetten (Agent Lightning)**

Veronderstel Python 3.10+ en een virtuele omgeving.

**1. Installatie van Vereisten**

Installeer bibliotheken:

bash

pip install agent-lightning *# Of clone van GitHub*

pip install torch gymnasium *# Voor RL*

pip install openai anthropic *# Voor LLM's*

pip install torch==2.7.0 torchvision==0.22.0 torchaudio==2.7.0 --index-url https://download.pytorch.org/whl/cu128

pip install flash-attn --no-build-isolation

pip install vllm==0.9.2

pip install verl==0.5.0

pip install "autogen-agentchat" "autogen-ext[openai]"

pip install langgraph "langchain[openai]" langchain-community

Stel API-sleutels in: export OPENAI\_API\_KEY=uw-sleutel.

**2. Framework Configureren**

* **Lightning Server Inrichten (voor RL-training)**: De server beheert training, data en API.

python

 from agentlightning import LightningServer

server = LightningServer(

model\_path="gpt-4o", *# LLM-model*

rl\_algorithm="LightningRL", *# Hierarchische RL*

dataset\_path="data/train.jsonl" *# Trainingsdata*

)

server.start() *# Luistert op bijv. localhost:8000*

Gebruik environment variables zoals RAY\_ADDRESS=auto.

 **Lightning Client Inrichten (voor agent-uitvoering)**: Verzamelt data en communiceert met server.

python

 from agentlightning import LightningClient

client = LightningClient(

server\_url="http://localhost:8000",

agent\_function=je\_agent\_functie *# Zie MDP-definitie*

)

client.run()

 **Agents Definiëren als MDP's**: States: Context (input + geschiedenis). Actions: LLM-outputs. Rewards: Sturen orchestratie.

python

 from agentlightning import MDPTransition, Agent

def je\_agent\_functie(state):

prompt = f"Context: {state['history']}\nHuidige input: {state['input']}"

action = llm.generate(prompt) *# LLM-call*

reward = scorer.evaluate(action, state['ground\_truth'])

return MDPTransition(state=state, action=action, reward=reward)

client = LightningClient(..., agent\_function=je\_agent\_functie)

 **Multi-Agents Configureren**: Ondersteunt single-LLM (rollen) of multi-LLM, met MARL voor optimalisatie.

python

* from agentlightning import MultiAgentClient, MARLOptimizer
* agent1 = Agent(role="SQL Writer", model="gpt-4o")
* agent2 = Agent(role="SQL Rewriter", model="claude-3")
* def multi\_agent\_run(state):
* action1 = agent1.act(state)
* updated\_state = {"context": state['context'] + action1}
* action2 = agent2.act(updated\_state)
* reward = scorer.multi\_evaluate([action1, action2], state['ground\_truth'])
* return [MDPTransition(state, action1, reward), MDPTransition(updated\_state, action2, reward)]
* client = MultiAgentClient(
* server\_url="...",
* agent\_function=multi\_agent\_run,
* optimizer=MARLOptimizer(algorithm="LightningRL")
* )

**3. Geheugen en Context Opbouwen**

Gebruik unified data interface: Trajecten bouwen geheugen op via states. Integreer observability (bijv. OpenTelemetry).

**4. Multi-Agent en RL-Orchestratie Implementeren**

Voorbeeld basis setup:

python

import os

from agent\_lightning import Client, Resource, Task

from uw\_agent import agent\_function

from environments import UwOmgeving

client = Client(os.environ["AgentLightningServerUrl"])

client.upload\_data("data/train.jsonl")

def agent\_run(resource: Resource, task: Task):

agent1 = agent\_function(resource.model\_api, "Rol1", model="gpt-4o")

agent2 = agent\_function(resource.model\_api, "Rol2", model="claude-3")

state = {"context": task.input, "geheugen": []}

for step in range(max\_steps):

action1 = agent1.act(state)

action2 = agent2.act(state)

reward = UwOmgeving.score(action1, action2, task.ground\_truth)

state["geheugen"].append((action1, action2, reward))

return reward

client.train(agent\_run, nworkers=4)

**5. Testen en Uitvoeren**

Voer client uit, monitor rewards, pas hyperparameters aan.

**6. Integratie met Andere Frameworks**

* Agno: Voeg Memory-drivers toe.
* LangGraph: Bouw workflows met custom nodes voor RL.

**Overwegingen en Beperkingen**

* Compatibiliteit: Test op API's.
* Resources: Begin klein, schaal met cloud.
* Raadpleeg documentatie voor updates.

Bovenkant formulier

Onderkant formulier