

Confrontare modelli LLM locali: benchmark per coding e funzioni agentiche

Tag: Agentic · AI · Benchmark · Coding · Guide · LLM

Ultimo aggiornamento: 2026-10-06

Confrontare modelli LLM locali: benchmark per coding e funzioni agentiche

Guida operativa per testare in locale i modelli che usi, misurando soprattutto le capacità di coding e funzioni agentiche (tool calling, uso di terminal, esecuzione multi-passo). Ogni strumento è spiegato: a cosa serve, come si installa e come si lancia il test, con i comandi completi.

0. Principio generale: serve un endpoint OpenAI-compatibile

Praticamente tutti i benchmark (e gli agenti) parlano con il modello tramite un'API OpenAI-compatibile (endpoint `/v1/chat/completions`, `/v1/models`). Il primo passo è sempre: servire il tuo modello locale dietro un'API di questo tipo. I due modi standard:

- vLLM — il migliore per throughput e supporto tool-calling; consigliato in generale.
- llama.cpp server — il più semplice per modelli GGUF e GPU consumer; supporta tool-calling con `--jinja`.

Se il modello è già servito da un altro servizio, salta questa sezione e usa direttamente la `BASE_URL`.

Regola d'oro per risultati confrontabili: stesso backend, stessa quantizzazione, stessa temperatura (per benchmark usa `temperature 0` o `0.1`). Se quantizzi Q4 e usi Q8 in produzione, i numeri non sono confrontabili.

1. Servire il modello con vLLM

A cosa serve

vLLM espone il modello locale come API OpenAI-compatibile, con supporto nativo a function/tool calling (usato da SWE-bench, BFCL, τ -bench, Terminal-Bench). È lo standard per i benchmark seri.

Installazione

```
# Python 3.9-3.12, GPU con driver CUDA (o backend CPU/rocm)
pip install vllm
```

Per GPU AMD: `pip install vllm --extra-index-url https://wheels.vllm.ai/rocm`

Avvio

```
vllm serve /percorso/modello \
  --port 8000 \
  --served-model-name mio-modello \
  --tensor-parallel-size 1 \
  --max-model-len 8192
```

- `--served-model-name`: nome usato nei `model=` dei benchmark (arbitrario).
- `--max-model-len`: contesto massimo; alzalo per task agentic lunghi (es. 16384).

- `--tensor-parallel-size N`: usa N GPU.

Verifica

```
curl http://localhost:8000/v1/models
```

2. Servire il modello con llama.cpp server

A cosa serve

Più leggero di vLLM, ideale per modelli già in formato GGUF e GPU consumer. Espone API OpenAI-compatible. Per i tool calling deve partire con `--jinja`, altrimenti i tool non vengono riconosciuti.

Installazione

Scarica il binary `llama-server` dalla release GitHub `ggerganov/llama.cpp` (cartella `build/bin/`), oppure compila:

```
git clone https://github.com/ggml-org/llama.cpp
cd llama.cpp
cmake -B build -DGGML_CUDA=on      # -DGGML_CUDA=on per NVIDIA, -DGGML_VULKAN=on per AMD
cmake --build build --config Release -j
# binary: build/bin/llama-server
```

Avvio

```
./llama-server \
-m /percorso/modello.gguf \
--port 8080 \
--jinja \
-c 8192 \
--alias mio-modello
```

- `--jinja`: fondamentale per il tool/function calling (usa il template `chat_template.jinja` del GGUF).
- `-c 8192`: contesto; alzalo per task lunghi.
- `--alias`: nome mostrato nell'API (default: hash del file).
- `--ctx-size` è sinonimo di `-c` in alcune versioni.

Verifica

```
curl http://localhost:8080/v1/models
```

Nota: `BASE_URL` per vLLM = `http://localhost:8000/v1`; per llama.cpp = `http://localhost:8080/v1`. In ogni esempio che segue, sostituisci `http://localhost:8000/v1` con la tua `BASE_URL`.

3. Aider polyglot (coding, screening rapido)

A cosa serve

Benchmark di coding multi-lingua su problemi reali (Python, C++, JS, Go, ...). È il filtro veloce: in pochi minuti vedi se un modello scrive codice che passa i test. Ottimo primo passaggio per scartare i modelli deboli.

Installazione

```
pip install aider-chat
```

Come funziona

Aider ha una modalità `--bench` che usa un messaggio-benchmark e misura quante volte il modello corregge il codice fino a far passare i test. I file di benchmark polyglot sono nel repo, sotto `aider/benchmarks/polyglot/` (es. `python.txt`, `cpp.txt`, `js.txt`, `go.txt`).

Test

Scarica i file polyglot e lancia per linguaggio:

```
# una tantum: porta i file polyglot locali
mkdir -p ~/aider-bench && cd ~/aider-bench
git clone --depth 1 https://github.com/Aider-AI/aider
cp aider/benchmarks/polyglot/*.txt .
```

Per ogni linguaggio:

```
python -m aider \
  --model openai/mio-modello \
  --base-url http://localhost:8000/v1 \
  --api-key dummy \
  --no-diff --no-cache \
  --bench \
  --message-file python.txt \
  --bench-report report-python.md \
  --print-diff
```

Sostituisci `python.txt` con `cpp.txt`, `js.txt`, ecc. per gli altri linguaggi.

Come leggere il risultato

Il report mostra per ogni problema: pass/fail e quante iterazioni ha servito. Più problemi superati e meno iterazioni = modello migliore. Confronta il total per linguaggio tra i tuoi candidati.

I flag esatti variano leggermente tra versioni di aider: in caso di dubbio, `python -m aider --help | grep -A2 -i bench`.

4. SWE-bench Verified (gold standard per agenti coding)

A cosa serve

Il benchmark più citato per agenti coding: il modello deve risolvere issue reali su repository Python reali (dare la patch che fa passare i test). È quello che predice meglio le prestazioni agentiche reali di coding. Pesante: 2295 istanze, richiede Docker e ore (anche un giorno) per run completo. Usalo solo per i finalisti; per un segnale rapido parti con una subset.

Installazione

```
# harness di valutazione
pip install swebench
# agente (SWE-agent): scarica il repo e installa
git clone https://github.com/SWE-agent/SWE-agent && cd SWE-agent
pip install -e .
```

1. Scarica i dati

```
python -m swebench.download_data -d verified -p ./swebench-data
# oppure dal HF hub: swebench/verified (istanze JSONL)
```

2. Configura SWE-agent verso il tuo endpoint

Crea `model_config.yaml`:

```
model:
  name: openai/mio-modello
  config:
    base_url: http://localhost:8000/v1
```

```
api_key: dummy
temperature: 0
```

Nota: i nomi esatti dei campi config variano leggermente tra versioni di SWE-agent (alcune usano `base_url`, altre `model_kwargs`). In caso di errore, consulta `swe-agent --help` e la doc "Model Configs" del repo.

3. Esegui l'agente sulle istanze

```
# subset rapida (es. 20 istanze) per un segnale
python -m swebench.run_agent \
  --model openai/mio-modello \
  --config model_config.yaml \
  --specification_path ./swebench-data/verified.jsonl \
  --run_id run1 --k 1
```

Per il run completo, rimuovi il limite di istanze. Per un segnale rapido, tieni la subset (es. 20-50 istanze).

4. Valuta le patch

```
python -m swebench.run_evaluation \
  --predictions_dir ./predictions/run1 \
  --experiment_name run1 \
  --k 1
```

Come leggere il risultato

Il report dà la % di issue risolte (resolved) e `resolved@1`. Per confronto, la stessa subset sugli stessi candidati: chi risolve più issue vince. I punteggi assoluti su SWE-bench sono bassi per modelli locali; conta soprattutto il ranking relativo tra i tuoi modelli.

È il test più costoso della lista. Non usarlo per lo screening: usalo dopo che gli altri test hanno selezionato 2-3 candidati.

5. Terminal-Bench (agenti su terminal)

A cosa serve

Task agentic eseguiti in un terminal dentro Docker: debug, script, pipeline, uso di CLI, lettura/writing di file, esecuzione multi-passo. Molto rappresentativo di un "agente da terminal" (come gli LLM CLI agent). Se usi intensivamente terminal/CLI, questo è il test più pertinente dopo SWE-bench.

Prerequisiti

Docker (gli task girano in container isolati).

Installazione

```
git clone https://github.com/laude-institute/terminal-bench && cd terminal-bench
pip install -e .
```

Configurazione verso il tuo modello

Terminal-Bench usa un file di configurazione YAML che dichiara il modello e l'agente. La sintassi esatta varia tra versioni: apri `config.yaml` di esempio nel repo e imposta:

- il modello (nome) e la `base_url` del tuo endpoint;
- `api_key` (anche dummy se il server locale non la valida);
- l'agente (es. `terminus`, o un tuo wrapper).

Schema tipico (verifica sul README della versione installata):

```
agent: terminus
config:
  - name: model
    value: openai/mio-modello
  - name: api_key
    value: dummy
  - name: base_url
    value: http://localhost:8000/v1
```

Esecuzione

```
# tutti i task (lento: ogni task ha un container Docker + timeout)
tbench run -c config.yaml

# subset rapida per un primo segnale
tbench run -c config.yaml --tasks task-1 task-2 task-3
```

Come leggere il risultato

Per ogni task: pass/fail (il modello ha completato l'obiettivo entro il budget di step/tempo). Il punteggio aggregato è la % di task completati. Confronta la % tra i tuoi candidati sulla stessa subset.

Terminal-Bench è lento (Docker per task). Parte con 5-10 task per confrontare, e completa solo per i finalisti.

6. BFCL — Berkeley Function Calling Leaderboard (tool calling)

A cosa serve

Misura la capacità di function calling / tool use: dato un contesto e una lista di funzioni, il modello deve chiamare la funzione giusta con gli argomenti corretti. È leggero e veloce (minuti, non Docker): perfetto per lo screening dei modelli che devi usare come agente con tool.

Installazione

```
git clone https://github.com/shreya-shipra/BerkeleyFunctionCallingLeaderboard && cd
BerkeleyFunctionCallingLeaderboard
pip install -r requirements.txt
```

Configura il tuo modello

Nel repo c'è `llm_configs.py` (o un file config equivalente, a seconda della versione): aggiungi una voce per il tuo endpoint OpenAI-compatibile:

```
MIO_MODELLO = {
    "name": "mio-modello",
    "model_name": "mio-modello",
    "api_key": "dummy",
    "base_url": "http://localhost:8000/v1",
}
```

Verifica sul README della versione clonata il nome esatto del file e delle chiavi (alcune versioni usano `llm_configs.py`, altre `model_configs`).

Esecuzione

```
python main.py --model mio-modello
```

Il benchmark gira più categorie (irrelevant-instruction, simple, parallel, multiple, etc.) e produce un report di accoppiamento (retrieval + matching) per categoria e un totale.

Come leggere il risultato

Il punteggio per categoria misura quante chiamate sono state formate e abbinate correttamente alle attese. Più alto = miglior tool calling. Usa la subset multi-turn + parallel per una visione più

"agentic".

Il punto debole del BFCL: misura "sa chiamare la funzione giusta", non "sa orchestrare più step". Per quello serve τ -bench o Terminal-Bench.

7. τ -bench (agente + utente simulato + tool)

A cosa serve

Un agente (il tuo modello) deve completare task in domini come retail, airline, telecom, parlando con un utente simulato e usando tool/API sotto vincoli di policy. Misura l'orchestrazione multi-step con tool: il passo che il BFCL non copre. Costo contenuto (minuti).

Installazione

```
git clone https://github.com/sierra-research/tau-bench && cd tau-bench
pip install -e .
```

Esecuzione

τ -bench usa due modelli: l'agente (il tuo, via OpenAI-compatibile) e l'utente simulato (puoi usare un modello hostato o un altro locale). Esecuzione tipica:

```
python -m tau_bench \
  --agent openai \
  --model mio-modello \
  --api_key dummy \
  --base_url http://localhost:8000/v1 \
  --domain retail \
  --user-model openai/gpt-4o \
  --user-api_key $OPENAI_API_KEY
```

Nota: i flag esatti variano tra versioni (alcune versioni usano un file di configurazione o un CLI `tau-bench run`). Consulta il README della versione clonata per la sintassi corrente; le chiavi sono le stesse: `model`, `api_key`, `base_url` per l'agente e `user-*` per l'utente.

Come leggere il risultato

Per ogni task il reward è 1 se l'agente ha raggiunto lo stato finale corretto e rispettato la policy, 0 altrimenti. Il punteggio aggregato è la % di task superati (e pass^k , la % che supera k run consecutivi). Confronta la % tra i tuoi candidati sullo stesso dominio.

Per test solo locale, puoi anche puntare l'utente simulato a un modello locale: in tal modo tutta la catena è locale.

8. LiveCodeBench (coding recente, anti-contaminazione)

A cosa serve

Problemi di programmazione (stile competitive programming) presi da gare recenti, con una finestra temporale per ridurre il rischio che il modello li abbia già visti in training. Misura capacità di problem solving + writing di codice che passa i test. Veloce rispetto a SWE-bench.

Installazione

```
git clone https://github.com/LiveCodeBench/LiveCodeBench && cd LiveCodeBench
pip install -r requirements.txt
```

Esecuzione

Il repo include uno script di valutazione che accetta un modello OpenAI-compatibile. Schema tipico:

```
python run.py \
```

```
--model_name mio-modello \  
--api_key dummy \  
--base_url http://localhost:8000/v1 \  
--eval_name evaluation \  
--problem_start 0 --problem_end 200
```

Verifica sul README della versione clonata i flag esatti; le chiavi sono sempre `model_name`, `api_key`, `base_url`, e i limiti di problemi da valutare.

Come leggere il risultato

Un accuracy su problemi risolti correttamente (`pass@1`). Più alto = migliore. Confronta la stessa finestra di problemi tra i candidati.

9. BigCodeBench (code gen + uso di library reali)

A cosa serve

Code generation dove la soluzione deve usare funzioni/operazioni reali (da library Python e C++): più vicino al coding reale che a problemi da gara. 521 task.

Installazione

```
git clone https://github.com/bigcode/BigCodeBench && cd BigCodeBench  
pip install -r requirements.txt
```

Esecuzione

BigCodeBench valuta tramite l'harness `bigcode-evaluation-harness`. Dopo aver generato le risposte (lo script di eval lo fa tramite API), lancia:

```
python -m bigcodebench.evaluation.evaluate \  
--model_name mio-modello \  
--api_key dummy \  
--base_url http://localhost:8000/v1
```

Verifica sul README della versione clonata il comando e le chiavi correnti; il principio è: genera le risposte sul tuo endpoint, poi valuta il `pass@k` sui test.

Come leggere il risultato

Un accuracy aggregato (`pass@1/pass@k`) per Python e C++. Confronta tra i candidati.

10. EvalPlus / HumanEval+ (code completion + test estesi)

A cosa serve

Versioni estese di HumanEval/MBPP con molti più test (molti casi d'errore e edge case) rispetto all'originale: più sensibile alla qualità del codice. Veloce, ottimo come controllo di base.

Installazione

```
pip install evalplus
```

Esecuzione

Due passi: genera le risposte, poi le valuta.

```
# 1. genera le soluzioni sul tuo endpoint  
python -m evalplus.evaluate --model_name mio-modello \  
--api_key dummy --base_url http://localhost:8000/v1  
# 2. valuta sui test estesi  
python -m evalplus.evaluation --model_name mio-modello
```

Nota: la sintassi per un endpoint OpenAI-compatibile varia tra versioni (alcune usano env OPENAI_BASE_URL/OPENAI_API_KEY); verifica con `python -m evalplus.evaluate --help`.

Come leggere il risultato

Un accuracy pass@1 per HumanEval+ e MBPP+. Baseline di riferimento: i modelli top 2025+ sono sopra 85-90 su HumanEval+. Confronta tra i candidati.

11. lm-evaluation-harness (IFEval, MMLU, GSM8K, HumanEval)

A cosa serve

Lo umbrella standard (EleutherAI) per runnare tanti benchmark generalisti in un solo comando: IFEval (instruction following — fondamentale per agenti), MMLU (conoscenza), GSM8K (ragionamento matematico), HumanEval (coding). Prompting standardizzato: evita numeri gonfiati da prompting fatto a casa.

Installazione

```
pip install lm-eval
```

Esecuzione (verso il tuo endpoint OpenAI-compatibile)

```
lm_eval \
  --model openai_completions \
  --model_args model=mio-modello,base_url=http://localhost:8000/v1,api_key=dummy \
  --tasks ifeval,mmlu,gsm8k \
  --batch_size 8 \
  --output_path ./results
```

Dove:

- `openai_completions` è il modello-wrapper per endpoint chat OpenAI-compatibili (vLLM/llama.cpp). Per endpoint /completions puri usa `openai`.
- `model=` è il `--served-model-name` (o `--alias`) che hai dato al server.
- `--tasks`: elenco separato da virgola. Per IFEval strict usa `ifeval` (alcune versioni hanno varianti `ifeval_strict`).
- `--batch_size`: alzalo se il server regge più richieste in parallelo.

Come leggere il risultato

I JSON in `./results` contengono la metrica per task (accuracy per MMLU/GSM8K, prompt-accuracy per IFEval). Per gli agenti: IFEval è il segnale più importante — un modello che non segue le istruzioni fallisce anche con i tool perfetti.

IFEval è leggerissimo: da girare sempre, per tutti i modelli, come primo screening.

12. Workflow consigliato (da screening a finalisti)

Ordina i test per costo, così non bruci ore sui modelli che scarti presto.

Fase 1 — Screening (minuti/pochi minuti per modello)

- 1 IFEval (lm-eval) — segue le istruzioni? Se no, scarta per uso agentico.
- 2 BFCL (subset multi-turn + parallel) — fa i tool calling giusti?
- 3 EvalPlus / HumanEval+ — baseline coding veloce.

4 Aider polyglot — coding multi-lingua pratico.

I modelli che non passano la fase 1 non proseguono.

Fase 2 — Profondità coding (ore, per 2-4 candidati)

- 1 LiveCodeBench — problem solving recente.
- 2 BigCodeBench — coding con library reali.
- 3 Terminal-Bench (5-10 task) — se usi molto terminal/CLI.

Fase 3 — Gold standard (ore/giornate, solo finalisti)

- 1 SWE-bench Verified — il predittore più affidabile per coding agentic.
- 2 τ -bench — orchestrazione multi-step con tool + utente simulato.

Tabella riassuntiva

Test	Cosa misura	Costo	Fase
IFEval (Im-eval)	instruction following	minuti	1
BFCL	tool/function calling	minuti	1
EvalPlus	coding + test estesi	minuti	1
Aider polyglot	coding multi-lingua	minuti	1
LiveCodeBench	problem solving recente	ore	2
BigCodeBench	coding + library reali	ore	2
Terminal-Bench	agente su terminal (Docker)	ore	2
SWE-bench Verified	risolvere issue reali (Docker)	ore/giorni	3
τ -bench	orchestrazione + tool + utente	minuti/ore	3

13. Note pratiche

- Backend e quantizzazione identici per tutti i modelli: vLLM per tutti, stessa quant GGUF o stesso formato, stessa temperature (0 per benchmark). Altrimenti i numeri non sono confrontabili.
- Stesso BASE_URL pattern: sostituisci `http://localhost:8000/v1` con l'endpoint del server corrente. Per vLLM è 8000, per llama.cpp 8080.
- Contesto: i task agentic (SWE-bench, Terminal-Bench, τ -bench) possono consumare molto contesto. Imposta `--max-model-len` / `-c` sufficiente (16k+), altrimenti i modelli falliscono per truncation, non per incapacità.
- Docker: SWE-bench e Terminal-Bench richiedono Docker funzionante. Verifica `docker run hello-world` prima.

- Costo GPU: SWE-bench completo è il più caro. Parti sempre dalle subset (20-50 istanze) per un confronto rapido e affidabile tra 2-4 candidati.
- Contaminazione: per coding "vero" preferisci SWE-bench Verified e LiveCodeBench (finestra recente) rispetto a HumanEval, che molti modelli hanno già visto.
- Versioni degli strumenti: le sintassi CLI cambiano tra versioni. Se un comando non va, consulta `--help` e il README della versione installata: le chiavi logiche (modello, api_key, base_url, task, subset) restano sempre le stesse.

14. Dove trovare tutto

Strumento	Repo
vLLM	github.com/vllm-project/vllm
llama.cpp	github.com/ggml-org/llama.cpp
Aider	github.com/Aider-AI/aider
SWE-bench	github.com/swe-bench/SWE-bench
SWE-agent	github.com/SWE-agent/SWE-agent
Terminal-Bench	github.com/laude-institute/terminal-bench
BFCL	github.com/shreya-shipra/BerkeleyFunctionCallingLeaderboard
τ -bench	github.com/sierra-research/tau-bench
LiveCodeBench	github.com/LiveCodeBench/LiveCodeBench
BigCodeBench	github.com/bigcode/BigCodeBench
EvalPlus	pypi.org/project/evalplus
lm-evaluation-harness	github.com/EleutherAI/lm-evaluation-harness