Maggio 2026

Analisi malware agentica: esperienze e metriche sul campo

Dalla sostenibilità dei token cloud alla sfida dei modelli locali

AGID | Agenzia per l'Italia Digitale

CERT-AGID

Analisi malware agentica: esperienze e metriche sul campo

Dalla sostenibilità dei token cloud alla sfida dei modelli locali

Maggio 2026

Abstract

L'impiego di agenti autonomi nella *malware analysis* si scontra con un limite strutturale: la crescita progressiva del contesto nelle sessioni lunghe e iterative. Per mantenere la coerenza tra i diversi passaggi, orchestrare gli strumenti e decodificare artefatti multilivello, il sistema satura rapidamente la propria memoria di lavoro. Questo rende l'affidamento esclusivo ai modelli cloud costoso, complesso e difficilmente sostenibile per CERT e SOC.

Questo lavoro valuta la fattibilità del trasferimento di tali workflow su infrastrutture interamente locali. Lo spunto nasce da un caso reale del CERT-AGID, in cui una *skill* eseguita tramite **Codex CLI di OpenAI**, basata su modello di frontiera **gpt-5.4**, ha ricostruito in totale autonomia un'analisi multi-stage complessa, arrivando a mappare l'ultimo stadio dell'infezione. Da quell'esperienza è stata avviata una campagna di test locali articolata su tre direttrici:

1. *benchmark* comparativi su **Codex CLI** + Ollama;
2. verifiche di robustezza del *runtime* su un client sviluppato internamente *modelcli* + *llama-server*;
3. un'analisi qualitativa su **Claude** + Ollama.

La tesi centrale è che, nei workflow agentici locali dedicati alla *malware analysis*, il degrado operativo emerga per limiti di infrastruttura, orchestrazione e *tool-use* prima ancora che per carenze intrinseche nel *reasoning* del modello. I dati indicano che la combinazione **Codex CLI + Ollama** (con *qwen3-coder:30b*) offre oggi il miglior compromesso per analisi rapide su task singoli; lo stack **modelcli + llama-server** garantisce invece il massimo controllo sul serving GGUF e sulla stabilità del ciclo locale. Il test su **Claude + Ollama** evidenzia infine come la gestione dei tool e del contesto da parte dell'orchestratore incida in modo diretto sulla tenuta complessiva della catena analitica.

Introduzione

I *Large Language Model* hanno superato il paradigma della semplice interfaccia conversazionale per trasformarsi in vere e proprie componenti operative. Nei contesti investigativi e di cyber security, i modelli non si limitano più a sintetizzare informazioni. Vengono usati come motori agentici capaci di leggere codice, eseguire strumenti da terminale, gestire una memoria di lavoro e sostenere un reasoning multi-step lungo sessioni complesse. Si tratta di un grande salto di qualità per la *malware analysis*, dove l'obiettivo principale non è la velocità della singola risposta, ma la capacità di collegare artefatti eterogenei e ricostruire l'intera dinamica di una minaccia.

Questa evoluzione introduce però un vincolo ingegneristico spesso sottovalutato: **il consumo cumulativo di token.** Un agente non lavora nel vuoto, a ogni passaggio deve registrare nel contesto gli output dei tool eseguiti, il codice analizzato, i log intermedi e i riassunti di stato. Per non perdere il filo dell'indagine, il sistema è costretto a rileggere una porzione sempre più ampia del passato a ogni nuovo turno di interazione. Nelle sessioni lunghe, questa dinamica accelera il degrado del workflow, innescando nei modelli locali stalli, loop ripetitivi o fallimenti nell'uso dei tool prima ancora che emergano i limiti reali di ragionamento del modello.

Nella triage e nel reverse engineering il problema è amplificato dalla natura stessa delle minacce moderne. Un malware raramente è un singolo file lineare. Più spesso si presenta come una catena multi-stage in cui, ad esempio, un documento Office malevolo nasconde stream OLE, che a loro volta richiamano script HTA, i quali eseguono codice PowerShell offuscato per caricare in memoria loader .NET e payload finali. Ogni livello di questa catena genera nuovi file e nuovi dati che devono rimanere coerenti all'interno dello stesso perimetro investigativo.

Il punto di partenza di questa ricerca è un'esperienza pratica sviluppata nell'ambito delle attività del CERT-AGID. In quell'occasione, una skill mirata ed eseguita tramite Codex CLI di OpenAI, con modello gpt-5.4, è riuscita a ricostruire autonomamente una catena multi-stage completa fino all'identificazione della tipologia di malware. Se da un lato questo successo ha confermato l'efficacia degli approcci agentici, dall'altro ha sollevato dubbi sulla sostenibilità a lungo termine di una dipendenza esclusiva dai frontier models in cloud, sia per i costi legati al volume dei token, sia per i vincoli di riservatezza imposti dal trattamento di campioni malware sensibili.

La domanda di fondo è pratica: **è possibile trasferire questi workflow complessi su infrastrutture interamente locali?** E quali sono i modelli, gli orchestratori e i runtime abbastanza maturi da sostenere task reali su macchine vincolate senza collassare sotto il peso del contesto o mostrare fragilità nell'uso dei tool?

Questo lavoro risponde osservando stack concreti su un hardware prosumer di riferimento. La campagna di test si articola su tre linee sperimentali: benchmark comparativi di modelli serviti tramite Ollama, analisi della robustezza del runtime locale su llama-server e verifiche qualitative sull'impatto di orchestratori alternativi.

Skill agentici per malware analysis

Per studiare questi comportamenti è utile partire dal concetto di *Agent Skill*. Una skill è, in sostanza, una directory contenente istruzioni, script e materiali di supporto che l'agente carica quando si attiva un determinato trigger. Il cuore di questo meccanismo è

tipicamente un file **SKILL.md**, che descrive quali strumenti usare, come organizzare il workflow e quali passaggi seguire.

Accanto al file principale possono comparire script bash o Python, decoder, template, riferimenti tecnici ed esempi. L'aspetto interessante è che le skill non richiedono infrastrutture particolarmente complesse: il modello legge le istruzioni e orchestra strumenti già disponibili nel sistema.

Nel contesto della malware analysis questo approccio è utile perché fornisce al modello una memoria operativa stabile:

- convenzioni investigative;
- decoder riutilizzabili;
- pipeline di estrazione;
- regole per l'organizzazione degli output;
- logiche di persistenza dell'analisi.

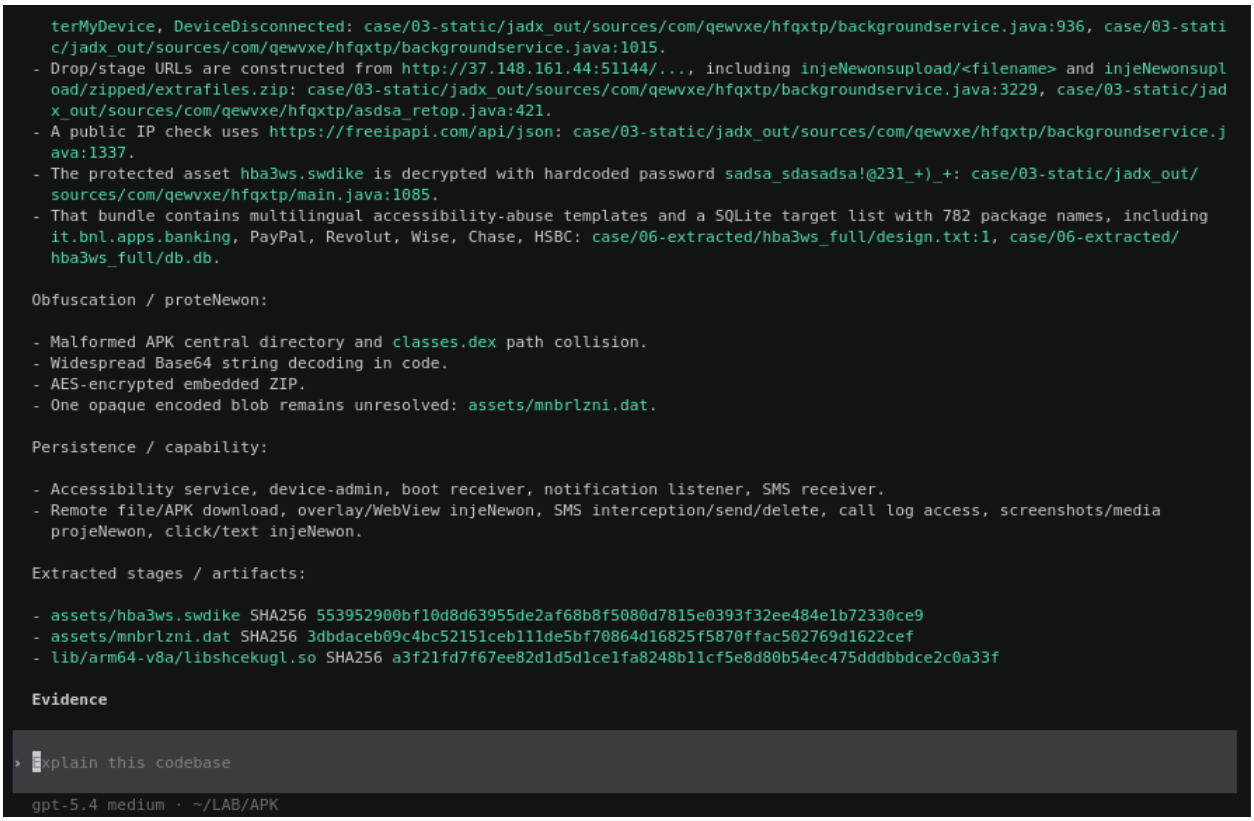
L'obiettivo non è automatizzare superficialmente il reverse engineering, ma ridurre la frammentazione del workflow mantenendo continuità lungo tutta la chain investigativa.

Caso reale: analisi automatizzata di un campione APK

Nell'ambito delle attività del CERT-AGID è stata sviluppata una skill[1] dedicata all'analisi statica automatizzata di malware. L'obiettivo dell'esperimento non era verificare il semplice riconoscimento di una firma o di una famiglia nota, ma osservare se un workflow agentico fosse in grado di superare tecniche attive di anti-analisi, sostenendo un'investigazione profonda senza interrompersi a causa del fallimento dei classici strumenti di parsing automatico.

Per il test è stato utilizzato **OpenAI Codex CLI** con il modello **gpt-5.4**, sfruttando le capacità di pianificazione del modello di frontiera associato alla sessione interattiva da terminale. Il campione analizzato era un pacchetto **APK bancario malevolo**, ingegnerizzato ad-hoc con tecniche di offuscamento strutturale per mettere in crisi i tool di reverse engineering: l'archivio ZIP presentava intestazioni corrotte, i parser standard fallivano la lettura e il file DEX principale era nascosto all'interno del container.

[1] Malware analysis skill for static malware analysis and reverse engineering across Windows, Linux, macOS, Android, Office documents, HTML/web payloads, scripts, source-code droppers, and multi-stage chains. https://github.com/guelfoweb/malware-analysis-static



Analisi del sample con Codex CLI + gpt-5.4 medium

L'aspetto più rilevante emerso dal test non è solo la classificazione finale del malware, ma il comportamento dinamico del workflow agentico:

- l'agente ha aggirato il parsing rotto estraendo direttamente il file DEX dal container tramite tecniche di carving;
- ha mappato la struttura interna del malware, individuando e analizzando le porzioni di codice distribuite su più file DEX e isolando gli asset cifrati;
- ha identificato ed esaminato i componenti nativi del pacchetto, estraendo le informazioni rilevanti dalle librerie compilate (.so).

Il sistema ha infatti estratto indicatori chiari relativi all'abuso dei servizi di accessibilità (*accessibility services*), alla predisposizione di schermate di overlay per il furto di credenziali su app bancarie target, all'uso di un protocollo MQTT per le comunicazioni con il C2 (tratto distintivo di *Copybara*) e alla capacità di scaricare ulteriori componenti dannosi da remoto.

Tuttavia, un ciclo investigativo così profondo genera una mole imponente di dati intermedi (file DEX estratti, dump di stringhe e log di decompilazione). Eseguire questo workflow sui modelli di frontiera in cloud non satura le loro capacità di calcolo, ma comporta un

consumo di token finanziariamente e computazionalmente insostenibile sul lungo periodo.

Da qui nasce la necessità di spostare il carico su modelli locali. È proprio in questo passaggio al locale, però, che emerge il vero problema: l'esplosione dei dati intermedi mette a durissima prova i runtime locali, saturando il contesto e portando il sistema al delirio operativo.

Metodologia dei benchmark locali

Per capire se workflow di questo tipo possano essere trasferiti in locale, è stata sviluppata una seconda linea di test articolata in tre piani distinti.

Il primo piano riguarda i benchmark comparativi su **Codex CLI + Ollama**, usati per osservare il comportamento di modelli diversi sul medesimo task e sullo stesso sample *JavaScript* offuscato.

I modelli per questa linea di test sono stati selezionati tra i più performanti nel panorama **open-weights** per il software development e scelti specificamente per il loro supporto nativo alla funzionalità di **tool-call**. In un workflow agentico di malware analysis, questa capacità è un prerequisito fondamentale, poiché consente al modello di tradurre il proprio ragionamento in comandi di terminale e invocare correttamente gli strumenti di analisi locali. In questa categoria rientrano, tra gli altri, *glm-4.7-flash*, *qwen3-coder:30b*, *qwen3-coder-next* e *gemma4:26b*.

Il secondo piano riguarda l'ingegneria del runtime locale. Per questa parte è stato utilizzato un client sperimentale dedicato, che abbiamo chiamato ***modelcli***, pensato per studiare stabilità del tool-use, sessioni e compattazione del contesto. Il modello di riferimento per questa linea è stato *Qwen3-30B-A3B-GGUF*, provato nella quantizzazione compatibile con il dispositivo disponibile e servito tramite ***llama-server*** con finestra contestuale estesa e offload GPU spinto quando sostenibile. In questo contesto *llama-server* è stato preferito come backend per i modelli GGUF perché consente un controllo più diretto dell'esecuzione e un adattamento più fine alla configurazione della macchina in uso, in particolare rispetto a contesto, offload GPU e footprint complessivo del modello.

Il terzo piano, più esplorativo, riguarda **Claude + Ollama**. Qui l'obiettivo non era ripetere un benchmark strettamente simmetrico a quello svolto con *Codex*, ma osservare se un orchestratore diverso producesse la stessa tendenza al deragliamento sul medesimo sample JS. In questa linea i modelli sopra citati sono stati eseguiti uno alla volta, con lo stesso prompt di analisi, lo stesso backend Ollama e la stessa telemetria hardware; i risultati vanno letti come osservazioni qualitative, perché la CLI di *Claude* in modalità

batch non espone la stessa osservabilità contestuale di *Codex* e tende a restituire l'output solo alla fine del run.

L'obiettivo era quello di osservare il comportamento dei modelli in condizioni vicine all'uso reale:

- analisi di file JavaScript fortemente offuscati;
- sessioni interattive lunghe;
- uso intensivo di tool locali;
- scrittura di artefatti intermedi;
- necessità di mantenere continuità di stato.

I problemi critici sono emersi quando il modello doveva alternare letture locali, parsing, compattazione del contesto e scrittura di risultati senza sforare la finestra di contesto o ripetere passaggi inutili. Per questo motivo i risultati vanno letti distinguendo chiaramente tra benchmark comparativi di modello, test di robustezza del runtime e osservazioni qualitative su un orchestratore alternativo. In pratica, le tre linee rispondono a domande diverse:

1. quale modello locale offre oggi il miglior equilibrio operativo immediato;
2. quale stack consente maggiore controllo e adattabilità quando l'obiettivo è costruire un workflow locale stabile su hardware limitato;
3. quanto il solo cambio di orchestratore possa modificare la persistenza della chain analitica a parità di backend locale.

Riproducibilità minima

I test locali sono stati eseguiti su un Dell Pro Max 16 Plus MB16250 con CPU Intel Core Ultra 7 265HX, 62 GB di RAM e GPU NVIDIA RTX PRO 5000 Blackwell Laptop con 24 GB di VRAM.

Il backend *Ollama* ha lavorato, nei run comparativi qui riportati, con context window reale di **32768 token**.

Per *modelcli* + *llama-server* il backend GGUF è stato configurato separatamente e non sempre con le stesse opzioni del ramo *Ollama*, perché l'obiettivo di quella linea era l'adattamento del serving locale più che la simmetria sperimentale.

I test sono stati eseguiti tramite run esplorativi singoli per modello, isolando la workstation da altri processi in background per non inquinare i dati di latenza e allocazione della VRAM.

Nei test batch, i blocchi operativi sono stati registrati esplicitamente come timeout configurato a 10 minuti. Parametri come temperatura, seed e contesti massimi hanno

ricalcato i preset di default dei rispettivi orchestratori, senza forzature artificiali; si è intervenuti manualmente sulle configurazioni di serving solo laddove necessario per prevenire il crash fisico del runtime locale.

Tassonomia degli esiti

Per ridurre l'ambiguità delle etichette qualitative usate nelle tabelle, in questo studio vengono adottate le seguenti categorie osservative:

- **Runtime stall:** il run non produce output utile entro il tempo fissato o resta dominato da condizioni di serving non sane, come offload CPU/GPU misto e footprint eccessivo.
- **Tool-use collapse:** il modello entra in loop, tool call malformati o ripetizioni che impediscono di avanzare nella chain analitica.
- **Semantic drift:** il modello abbandona il tracciamento semantico del sample e ricade in euristiche superficiali, pattern matching o IoC grepping.
- **Premature closure:** il modello chiude il task con un report formalmente concluso ma troppo generico o non supportato da ricostruzione concreta.
- **Best operational compromise:** il run non è perfetto, ma rappresenta il miglior equilibrio osservato tra footprint, progressione dell'analisi e controllabilità del workflow.

Le etichette sintetiche riportate nelle tabelle derivano da questa tassonomia e vanno intese come classificazioni osservative del comportamento del sistema, non come metriche universali.

Risultati e osservazioni

I risultati confermano un punto generale: il comportamento osservato dipende dallo stack nel suo complesso, non solo dal modello. Prompt, orchestratore, formato dei tool-call, gestione della sessione e osservabilità del backend pesano quanto la qualità intrinseca del reasoning.

Un client ad hoc con llama-server

L'obiettivo non è confrontare famiglie diverse di modelli ma adattare un modello GGUF specifico alla workstation disponibile. Un client scritto ad hoc *(modelcli)* appoggiato a *llama-server* si è rivelato un banco di prova utile: in questo assetto, il modello di riferimento è stato ***Qwen3-30B-A3B-GGUF***, usato per verificare la tenuta del ciclo agentico locale. I test su *llama.cpp* non compaiono nella tabella comparativa sotto riportata.

Il risultato principale non è stato un “primato” qualitativo del modello, ma il raggiungimento di un runtime sensibilmente più controllabile: sessioni isolate per directory, compattazione del contesto più affidabile, fallback locali quando il riassunto col modello falliva, limiti espliciti sui chunk letti e scritti, riduzione dei loop ripetitivi e migliore leggibilità degli errori di tool-use grazie al client scritto su misura.

Sul JS è riuscito a riconoscere la funzione che restituisce la tabella di stringhe e a isolare una serie di indicatori utili alla triage. In più, il modello è arrivato a evidenziare alcune espressioni concrete di costruzione o caricamento dinamico, come la concatenazione. Il punto rilevante non è che il modello abbia completato automaticamente una deoffuscazione perfetta, ma che, una volta stabilizzato il runtime, sia riuscito a produrre risultati parziali verificabili e progressivamente riutilizzabili in un workflow locale.

In sintesi, i risultati di *modelcli* + *llama-server* possono essere riassunti così:

- **Robustezza del runtime locale su GGUF:** ciclo agentico più controllabile su task assegnati singolarmente, compattazione più affidabile, errori di tool-use più leggibili.
- **Analisi del JS offuscato:** riconoscimento del pattern di offuscamento, string table, lookup per offset e rami di caricamento dinamico.

Codex con Ollama

Per rendere i risultati confrontabili è stato eseguito lo stesso prompt sullo stesso sample JS con quattro modelli diversi, tramite **Codex + Ollama** monitorando sessione, token e risorse con uno script creato per l’occasione con l’aiuto dell’IA. La tabella seguente sintetizza gli indicatori più rilevanti emersi durante questi run comparativi.

Modello	Backend Ollama	Picco VRAM	Picco RAM	Token/sessione osservati	Saturazione ctx reale	Comportamento osservato	Esito
glm-4.7-flash	100% GPU	21,7 / 24,5 GiB	4,1 GiB	~25.980	79,3%	Avvio rapido, poi script temporanei fragili e fallback a grep/pattern matching	Semantic Drift

qwen3-coder-next	56% CPU / 44% GPU	23,6 / 24,5 GiB	32,7 GiB	~12.032 iniziali	non significativo	Run dominato da offload misto CPU/GPU e footprint eccessivo; nessun tool-use in tempo utile	Runtime Stall
qwen3-coder:30b	100% GPU	21,1 / 24,5 GiB	4,6 GiB	~28.611	87,3%	Individua frammenti rilevanti, ma ricade su grep/rg e matching testuale	Best Operational Compromise
gemma4:26b	100% GPU	19,2 / 24,5 GiB	5,3 GiB	~25.034	76,4%	Avvio più ordinato, ma chiusura su rg e parsing Python fragile	Tool-use Collapse

Il confronto rende evidente che il problema non è solo "quale modello risponde meglio", ma **quale modello mantiene un equilibrio credibile tra footprint hardware, consumo di contesto e disciplina del tool-use.**

In questo test *gemma4:26b* è stato il più leggero sul piano della VRAM, mentre **qwen3-coder:30b** ha mostrato il miglior compromesso tra velocità di avvio, uso della GPU e capacità di avanzare concretamente nell'analisi, pur senza evitare il deragliamento finale verso euristiche troppo superficiali.

Il caso *qwen3-coder-next* indica un problema di configurazione, non un limite del modello. Il carico diviso tra CPU (56%) e GPU (44%), insieme ai 32 GB di RAM occupati, dimostra che il modello non è stato caricato interamente in scheda video per un errore di offload o di quantizzazione su *Ollama*. Questo run descrive il fallimento dell'infrastruttura locale in quel test specifico, non il valore reale del modello nel tool-use.

Se l'obiettivo è individuare la soluzione più adatta per analisi locali rapide e benchmark comparativi, la risposta che emerge dai dati - **nonostante nessun modello abbia completato l'analisi in modo efficiente** - è l'accoppiata **Codex + Ollama** con **qwen3-coder:30b**, configurazione dimostratasi la più equilibrata tra quelle osservate.

Se invece la domanda diventa quanto controllo si abbia sul runtime e quanto sia possibile adattare il serving di un modello GGUF alla macchina disponibile la risposta si sposta verso un **client orchestrator scritto ad hoc (modelcli) + llama-server**, che ha prodotto risultati analitici più parziali ma anche un ambiente di esecuzione più governabile.

Claude con Ollama

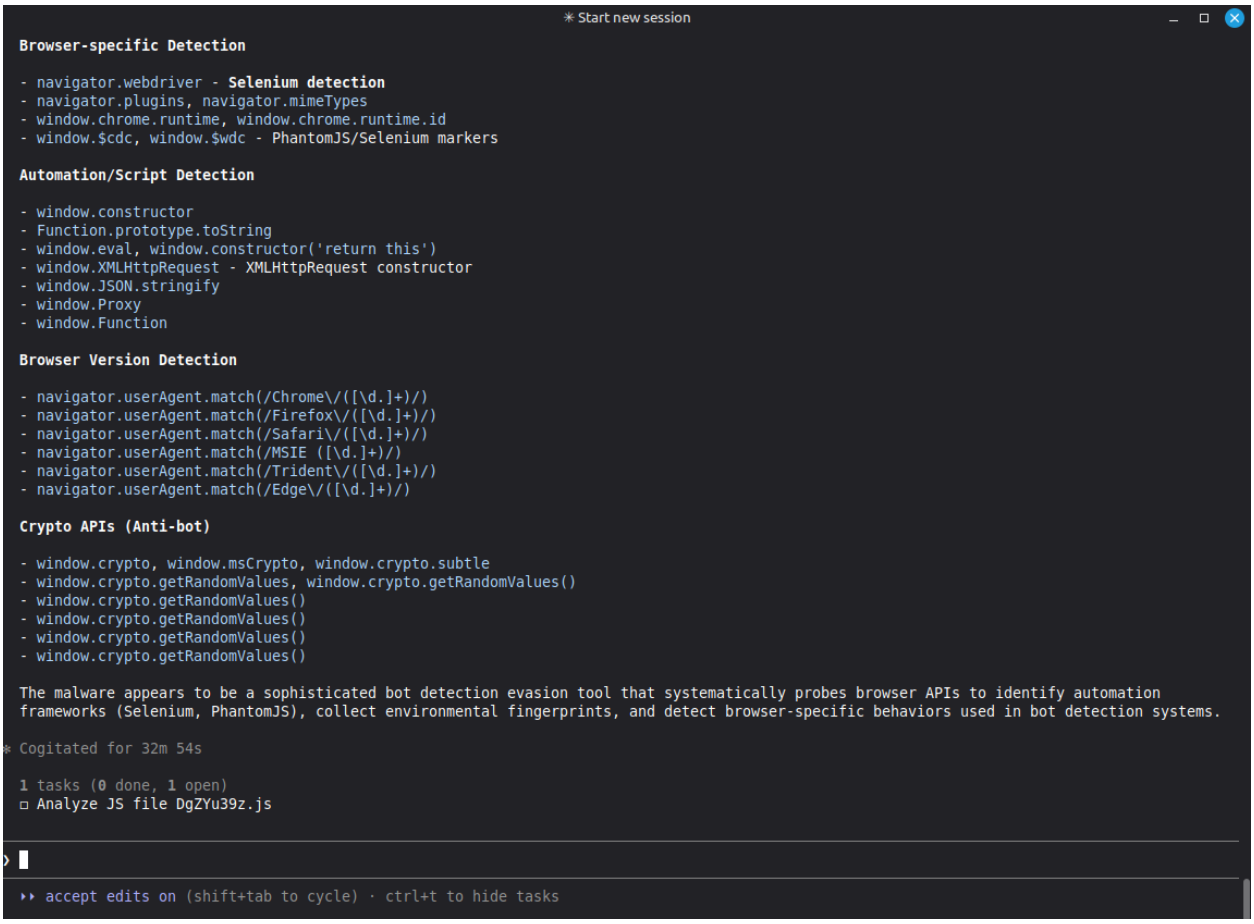
Una seconda osservazione emerge però cambiando orchestratore. Gli stessi modelli sono stati eseguiti anche con **Claude + Ollama**, mantenendo lo stesso sample e lo stesso prompt, ma abbandonando la telemetria contestuale ricca di *Codex* in favore di una misura più semplice fondata su tempo totale, footprint hardware e qualità dell'output finale. In questo assetto il comportamento cambia sensibilmente: i run restano stabili lato backend, ma la chiusura del task diventa molto più lenta e meno uniforme.

Modello	Tempo osservato	Picco VRAM	Picco RAM	Output finale	Esito
qwen3-coder:30b	277 s	21,1 / 24,5 GiB	5,7 GiB	report finale breve ma generico, senza ricostruzione concreta della chain	Premature Closure
gemma4:26b	> 480 s	19,2 / 24,5 GiB	6,5 GiB	nessun output finale nel batch test entro il timeout	Runtime Stall
qwen3-coder-next	> 360 s	23,6 / 24,5 GiB	33,6 GiB	nessun output finale nel batch test entro il timeout	Runtime Stall
glm-4.7-flash	> 600 s	21,7 / 24,5 GiB	5,6 GiB	nessun output finale nel batch test entro il timeout	Runtime Stall

Il dato interessante non è che **Claude + Ollama** vinca o perda contro **Codex + Ollama**, ma che sposti il punto di rottura. Nei run batch qui misurati *Claude* non ha mostrato i deragliamenti rapidi e rumorosi tipici di *Codex* sullo stesso sample: niente loop evidenti di grep, niente tool call fragili immediatamente visibili, niente ripetizione palese degli stessi passi.

Al tempo stesso, però, tre modelli su quattro non hanno restituito alcun output finale utile entro il timeout fissato a 10 minuti. In altre parole, *Codex* tende a fallire in modo prima osservabile e più facilmente classificabile, mentre *Claude* tende a mantenere il backend occupato più a lungo, lasciando aperta la possibilità di una chain analitica più profonda.

Questa differenza è rilevante anche perché una **sessione interattiva** separata, svolta sullo stesso sample con **Claude + Ollama + glm-4.7-flash**, ha prodotto una analisi parziale dopo circa 30 minuti senza i deragliamenti tipici dei run *Codex*.



Analisi del JS con Claude + Ollama + gml-4.7-flash

A parità di backend locale, la qualità della chain analitica percepita dipende in modo sostanziale anche dall'orchestratore e dal modo in cui il client gestisce tool, stato e tempi di chiusura del task.

Perché iniziare a guardare ai modelli locali

Nonostante questi limiti, il tema dei modelli locali è destinato a crescere di importanza. L'aumento dei costi dei workflow cloud rende più interessante eseguire almeno una parte delle attività su infrastrutture controllate localmente, soprattutto nei contesti CERT e SOC, dove i campioni e gli artefatti intermedi non sono sempre trasferibili verso servizi esterni.

Il punto non è più soltanto selezionare il modello con la risposta migliore. Negli scenari agentici la priorità si sposta sulla sostenibilità dell'intero workflow: persistenza del ragionamento, tenuta del contesto lungo, stabilità nei task iterativi, disciplina nel tool-use e qualità dell'orchestrazione. I benchmark su *Codex + Ollama*, le prove su *modelcli + llama-server* e i test con *Claude + Ollama* dimostrano che il confine tra un workflow locale affidabile e uno fragile dipende dal wrapper operativo almeno quanto dalle capacità intrinseche del modello.

Conclusioni

L'esperienza descritta mostra che un sistema agentico ben orchestrato può affrontare attività di malware analysis mantenendo continuità logica, correlazione tra artefatti e reasoning multi-stage. Il punto decisivo non è la sola velocità di risposta, ma la capacità del workflow di mantenere lo stato operativo durante task lunghi e iterativi.

I risultati suggeriscono che la vera metrica di analisi non è il modello isolato, ma lo stack completo. **Codex + Ollama** è oggi la soluzione più convincente quando servono benchmark rapidi, confrontabili e subito operativi, con **qwen3-coder:30b** come miglior compromesso osservato. **modelcli + llama-server**, al contrario, è meno brillante come esperienza immediata ma più forte quando l'obiettivo è controllare il serving GGUF, gestire meglio sessioni e compattazione, e rendere più robusto il ciclo agentico locale.

Claude + Ollama, infine, non ha ancora mostrato la stessa riproducibilità in batch a causa del timeout fissato a 600 secondi, ma lascia intravedere una superiore persistenza della chain analitica. Si è trattato infatti dell'unica configurazione in grado di completare parzialmente l'analisi una volta portata in modalità interattiva con il modello **glm-4.7-flash**.

I dati raccolti indicano chiaramente che per mitigare il degrado del contesto e l'instabilità dei runtime locali, l'approccio architetturale più efficace non prevede sessioni monolitiche e omnicomprensive, come per i modelli di frontiera eseguiti in cloud, ma l'esecuzione di task singoli, verticali e atomici, supportati da una gestione controllata e stringente di ogni singola invocazione dei tool.

La sfida, quindi, non consiste soltanto nel trovare il modello migliore, ma nel costruire sistemi agentici sostenibili, verificabili e abbastanza robusti da supportare attività operative reali nel lungo periodo. In quest'ottica, benchmark realistici, telemetria affidabile e disciplina nell'orchestrazione dei tool diventano parte integrante della valutazione tecnica.

Resta infine un rischio meno discusso ma ormai evidente. La rapidità e la fluidità dei modelli cloud di frontiera stanno cambiando il metro con cui giudichiamo gli strumenti locali. Un modello quantizzato che un anno fa sarebbe apparso come un ottimo "secondo analista" oggi può sembrare lento o cognitivamente faticoso solo perché il confronto implicito avviene con sistemi remoti molto più fluidi. Più ci abituiamo a delegare interpretazione, correlazione e decisioni operative ai modelli cloud, più rischiamo di perdere controllo diretto sul processo e di accettare come inevitabile una dipendenza strutturale dall'esterno. Per questo la valutazione dei modelli locali non dovrebbe limitarsi

alla produttività immediata, ma includere anche autonomia tecnica, osservabilità del processo e mantenimento delle competenze analitiche.