

The science behind the report:

Put agentic AI to work on familiar, general purpose servers

This document describes what we tested, how we tested, and what we found. To learn how these facts translate into real-world benefits, read the report Put agentic AI to work on familiar, general purpose servers.

We concluded our hands-on testing on October 14, 2026. During testing, we determined the appropriate hardware and software configurations and applied updates as they became available. The results in this report reflect configurations that we finalized on October 13, 2026, or earlier. Unavoidably, these configurations may not represent the latest versions available when this report appears.

Our results

To learn more about how we have calculated the wins in this report, go to https://facts.pt/calculating-and-highlighting-wins. Unless we state otherwise, we have followed the rules and principles we outline in that document.

We tested a Dell™ PowerEdge™ R770 server with two Intel® Xeon® 6767P processors. The processors support Intel® Advanced Matrix Extensions (Intel® AMX), and we used Intel AMX for inferencing during our testing. The server ran the OpenVINO™ Model Server (OVMS) inference engine with the Qwen3-8B large language model (LLM) at INT4 weight precision, and our proprietary agentic benchmark framework drove AI agents against it with the framework's financial-services workload.

We ran each measurement sweep from a saved control-panel queue. Each sweep first ran 28, 31, and 34 concurrent agents three times each, which placed the 30-second p95 target between 28 and 31 agents. The sweep then ran 29 and 30 agents three times each on its own to find the exact limit, for 15 runs in total. Each run's measurement window was 420 seconds. We ran the full sweep three times to check consistency, and all three sweeps found that 30 agents was the highest count under the target.

At each concurrency level, we report the run with the median tasks per minute of its three runs, and every value for that level comes from that run. Of the three sweeps, we report the one whose selected 30-agent run had the median tasks per minute (110.86). All results data below comes from that sweep.

Our headline throughput metric is steady-state throughput (tasks per minute): the tasks that completed inside the 420-second measurement window, divided by 7 minutes. We also report all-completed throughput, which counts every completed task in the run and divides by the total wall-clock time of the run, so it is lower than steady-state throughput, because the run's start and end add time when fewer tasks are finishing.

Table 1: Summary of results for the Dell PowerEdge R770 running OpenVINO Model Server with Qwen3-8B INT4 on the financial-services workload (moderate tier).

Metric	Result
Maximum concurrent agents meeting the 30-second p95 service level (M)	30
Steady-state throughput at M (tasks per minute)	110.86
Steady-state throughput at M (tasks per hour)	6,651.43
All-completed throughput at M (tasks per minute)	104.51
Median p95 task latency at M (s)	28.06
p95 task latency at M + 1, 31 agents (s)	30.39 (exceeds the service level)
Median (p50) task latency at M (s)	14.20
Output tokens per second at M (all agents combined)	215.5
Average task score at M (0 to 1)	0.948

Output tokens per second are the total number of tokens the model generated for all tasks that completed inside the 420-second measurement window, across all agents, divided by 420. It counts only the model's output, not the prompts it read. At 30 agents, tasks averaged about 117 output tokens each, because most answers are short JSON findings or decisions.

Table 2: Results by concurrency level in the reported sweep. We ran each level three times; each row shows the run with the median tasks per minute at that level.

Number of agents	Steady-state tasks per minute	p95 latency (s)	p50 latency (s)	Average score	Meets service level target
28	111.71	26.57	13.37	0.956	Yes
29	111.43	26.89	13.62	0.959	Yes
30	110.86	28.06	14.20	0.948	Yes
31	105.29	30.39	15.08	0.948	No
34	107.86	32.04	15.73	0.951	No

Table 3: Results by task type at 30 concurrent agents, from the selected 30-agent run. Task counts cover the tasks that completed inside the measurement window; latencies cover the tasks that started inside it, the same rule the service-level test uses.

Task type	Number of tasks completed	Share of tasks (%)	p50 latency (s)	p95 latency (s)
Transaction reconciliation	98	12.6	15.19	21.46
Portfolio valuation	115	14.8	13.98	14.87
Regulatory ratios	260	33.5	4.73	14.61
Peer benchmarking	72	9.3	21.98	23.79
Filing analysis	103	13.3	14.41	22.41
Fraud alert triage	77	9.9	19.13	28.10
Filing Q&A	36	4.6	28.84	31.70
Board-pack narrative	15	1.9	94.96	113.54
All task types	776	100.0	14.20	28.06

System configuration information

Table 4: Detailed information on the system we tested.

System configuration information	Dell PowerEdge R770
BIOS name and version	Dell 1.9.16
Operating system name and version/build number	Ubuntu® 26.04.1 LTS, kernel 7.0.0-38-generic
Date of last OS updates/patches applied	2026-10-10
Power management policy	BIOS system profile: Performance OS: See Table 5
Processor	
Number of processors	2
Vendor and model	Intel Xeon 6767P
Core count (per processor)	64
Core frequency (GHz)	2.40 base (Intel specification, Server Mode; 2.80 in the Intel Speed Select Compute Mode); 3.90 maximum turbo
Stepping	1 (family 6, model 173)
Microcode	0x1000434
Threads per core	2 (Hyper-Threading enabled)
Cache	48KB data and 64KB instruction L1 per core; 2MB L2 per core; 336MB L3 per processor
Intel AMX	Enabled and used by OVMS
Other instruction-set extensions	Intel® Advanced Vector Extensions 512 (Intel AVX-512) (including FP16, BF16, and VNNI), AVX-VNNI
NUMA nodes	2 (one per processor); sub-NUMA clustering off
Integrated accelerators	Intel DSA and Intel IAA present; not used by the workload
Thermal design power (W)	350 (Intel specification)
Memory module(s)	
Total memory in system (GB)	1,024
Number of memory modules	16 (of 32 slots; one module per memory channel)
Vendor and model	Micron® MTC40F2046S1RC64BD2
Size (GB)	64
Type	DDR5 RDIMM, dual rank, ECC
Speed (MT/s)	6,400
Speed running in the server (MT/s)	6,400
Storage controllers	
Storage controller 1: Vendor and model (boot)	Dell BOSS-N1, firmware 2.2.13.2038
Storage controller 2: Vendor and model (additional volume)	Dell PERC H365i Front, firmware 8.18.1.0.15-142

System configuration information	Dell PowerEdge R770
Local storage (OS)	
Number of drives	2
Drive vendor and model	Dell EC NVMe® ISE 7450 RI M.2 SSD
Filesystem	ext4
Local storage (data)	
Number of drives	4
Drive vendor and model	Dell DC NVMe CD8 U.2 1.92TB
Volume configuration	Linux® software RAID 10 (md, 512 KB chunk, two near copies), XFS, 3.5 TB usable
Volume use	Benchmark framework, results, and the Docker® data root, including the OVMS model volume, and the containerd root (image layers)
Local storage (additional)	
Number of drives	8
Drive vendor and model	Micron MTFDDAK3T8TGA-1B
Volume configuration	14TB hardware RAID virtual disk, XFS, RAID 10
Volume use	Model archive, not in the measured path
Network adapter	
Vendor and model	2 x Intel Ethernet Controller E610 (2.5GBASE-T); 1 x Intel Ethernet Controller E810-C (QSFP)
Number and type of ports	Each E610: 2 x 2.5GbE RJ45; E810-C: 2 x QSFP 100GbE
Driver version and firmware	E610: ixgbe (in-kernel driver, kernel 7.0.0-38), firmware 1.01 0x8000cde4 1.3768.0; E810-C: ice (in-kernel driver), firmware 4.80 0x8002069f 24.0.5.
Cooling fans	
Vendor and model	Dell Silver
Number of cooling fans	6
Power supplies	
Vendor and model	DELL PWR SPLY, DELTA
Number of power supplies	2
Wattage of each (W)	1,500
Systems management	
iDRAC firmware version	1.30.60.50

Table 5: Operating system and power-management settings we observed on the server under test.

Setting	Value
CPU frequency scaling driver	intel_cpufreq (intel_pstate in passive mode)
Scaling governor	schedutil on all 256 logical CPUs
Scaling minimum and maximum (MHz)	800 and 3,900
Turbo	Enabled
Energy performance bias	6 (balanced performance)
Energy performance preference	Not exposed in passive mode
Idle states (C-states)	POLL, C1, and C1E enabled; C6 and C6P disabled
Hyper-Threading	Enabled (2 threads per core, 256 logical CPUs)
Transparent huge pages	madvise (enabled and defrag)
Automatic NUMA balancing	On
Swap	8 GiB (8,388,604 kB); swappiness 60
Kernel command line	Ubuntu defaults

Table 6: Software we used.

Software	Version and details
Operating system	Ubuntu 26.04.1 LTS, kernel 7.0.0-38-generic
Docker Engine	29.8.2 (data root on the RAID 10 data volume)
Docker Compose	v5.6.0
Inference server	OpenVINO Model Server 2026.4.1 (build 2026.4.1.cfc9ac721); OpenVINO 2026.4.1-22982-e213a147257; OpenVINO GenAI 2026.4.1.0-3408-ab01d1771e4
Inference server image	openvino/model_server:2026.4.1 (CPU build), pinned by digest sha256:148e3fe802261ed0b5b634de3bbccdefe7176a218ec34a88e382b40704c34a1c
Large language model	OpenVINO/Qwen3-8B-int4-ov (Intel-published OpenVINO IR of Qwen3-8B with INT4 weights; 4.6 GB on disk)
Chat template	qwen3-8b-nothink.jinja from the framework (Qwen3 thinking mode off by default); SHA-256 b77cad51b858d081722e26e2b74fbc9043913140e2924df91ff2f334482b8c08
Benchmark framework	PT agentic benchmark framework, commit 19fb4cef1af7e5688ac7d64cfd302137f8210649
Agent framework	LangGraph 1.2.14, LangChain 1.4.4, langchain-core 1.6.10
Runner Python® and libraries	Python 3.11.17; OpenAI Python library 3.29.0; NumPy 2.4.6; pandas 3.0.6; NetworkX 3.6.1; asyncpg 0.32.0; Pydantic 2.14.1; HTTPX 0.28.1
Retrieval models (CPU)	Embedder BAAI/bge-small-en-v1.5 and reranker Xenova/ms-marco-MiniLM-L-6-v2, both on ONNX Runtime CPU execution
Database	PostgreSQL® 16.15 (postgres:16-alpine image, pinned by digest sha256:721873c34ceb9f8d8fc265984940dc982404c105f19ad51be9fdc5970a6080ea)
Monitoring (not in the measured path)	VictoriaMetrics v1.108.1, Grafana OSS 11.6.0
Host Python	3.14.4 (runs only the deployment script)

Table 7: OpenVINO Model Server parameters.

Parameter	Value
Target device	CPU
KV cache size (GB)	80
Maximum concurrent sequences	64
Prefix caching	On
Dynamic split fuse (prompt splitting)	Off, so the server reads a whole prompt of up to 32,768 tokens in one pass
Maximum batched tokens	32,768
Metrics endpoint	On
Context window	32,768 tokens, enforced by the benchmark client from each reply's token usage (OVMS has no context-window setting)
Generation settings	Maximum 6,144 output tokens per request; temperature 0
Container restart policy	Restart unless stopped

Table 8: Processor placement.

Component	Placement
OpenVINO Model Server	Socket 0: even CPU IDs 0 through 126, one logical CPU on each of the 64 physical cores; we left the 64 Hyper-Threading siblings on socket 0 idle
Agent runner (LangGraph agents, retrieval embedder and reranker, per-task analysis code)	Socket 1: all 128 logical CPUs (odd CPU IDs 1 through 255)
Control panel	Socket 1 (same CPUs as the runner)
PostgreSQL	Socket 1 (same CPUs as the runner)

About the workload

We used the financial-services workload of the PT agentic benchmark framework, which runs realistic AI-agent workflows end to end. An agent task is one self-contained piece of financial back-office work. The agent, built with LangGraph, pulls data from a PostgreSQL database of synthetic bank and market records, from PDF filings, or by searching a library of real U.S. Securities and Exchange Commission (SEC) filings. It runs ordinary analysis code on that data (Python, pandas, and SQL) and calls the large language model to write the answer: a narrative, a JSON list of findings, or a decision. The framework grades each answer automatically against a known answer key. Most task types make one model call after the data work; the fraud alert triage task makes up to nine calls in a tool-using loop. A task passes when its score meets the threshold for its task type.

The suite contains eight task types. Each concurrency level splits its agents as evenly as possible across the eight task types, and each group of agents repeats only its own task type, so every task type has about the same number of tasks in flight. Fast task types complete many more tasks in a window than slow ones.

- **Transaction reconciliation**
  - What the agent does: Replays each account's transactions for a batch of 200 accounts, compares computed and posted closing balances, records an exception for each mismatch, and writes an exceptions narrative
  - Data and tools: PostgreSQL account and transaction tables (500 branches of 100 accounts each, 1 million transactions over 7 days); writes an exceptions table; one model call
  - Score: F1 score over the flagged accounts
  - Passes when: F1 of 0.5 or higher

- **Portfolio valuation**
  - What the agent does: Marks one portfolio to market, computes net asset value (NAV) and the top three sector exposures, and writes a NAV narrative
  - Data and tools: PostgreSQL market data (1,000 tickers over 1,260 trading days), positions (40 portfolios, 100,000 positions), and company tables; pandas; one model call
  - Score: 0.8 x NAV accuracy (0.5 percent tolerance) + 0.2 x overlap of the top three sectors
  - Passes when: Score of 0.5 or higher
- **Regulatory ratios**
  - What the agent does: Computes regulatory ratios for 200 entities, checks each against the limits for its jurisdiction, and writes a compliance narrative
  - Data and tools: PostgreSQL ratio inputs (500 entities, 15 ratios each) and regulatory thresholds; ratio calculator and threshold checker tools; one model call
  - Score: F1 score over the reported breaches
  - Passes when: F1 of 0.5 or higher
- **Peer benchmarking**
  - What the agent does: Ranks the tickers in one sector by trailing return with SQL window functions and writes sector commentary
  - Data and tools: PostgreSQL market data joined with company data; SQL aggregate tool; one model call
  - Score: 0.5 x top-five overlap + 0.2 x bottom-five overlap + 0.3 x percent-rank accuracy
  - Passes when: Score of 0.5 or higher
- **Filing analysis**
  - What the agent does: Parses key financials from a batch of five synthetic annual-report (10-K) PDFs, looks up the same metrics in the database, flags differences of 5 percent or more, and writes the discrepancies as JSON with a risk summary
  - Data and tools: 500 synthetic 10-K PDFs, one per company (each task reads five); PostgreSQL historical financials; one model call
  - Score: F1 score over the flagged company and metric pairs
  - Passes when: F1 of 0.5 or higher
- **Fraud alert triage**
  - What the agent does: Takes one alert raised by three SQL fraud rules, investigates it with database tools in a reason-and-act loop, and decides whether to escalate or clear it; the corpus plants real fraud and benign look-alikes
  - Data and tools: PostgreSQL transactions (7-day window, 1 million rows); tools to profile accounts, look up an account, scan an account's transactions, check a rule, and run SQL aggregates; up to eight tool steps plus one forced final-decision turn
  - Score: 1 if the decision is correct, otherwise 0; a reply without a final decision counts as a format failure
  - Passes when: Score of 0.8 or higher (in effect, a correct decision)
- **Filing Q&A**
  - What the agent does: Answers 10 questions about synthetic 10-K filings in one JSON reply, using retrieval-augmented generation over a library of real SEC filings in which the answer-bearing synthetic filings are hidden
  - Data and tools: Hybrid retrieval (keyword BM25 search, dense embedding search, rank fusion, and reranking) returning three passages per question (up to 30 per task) from a 270,806-passage library; questions come from 11 templates: 9 transcription, 2 derived ratios; one model call
  - Score: Fraction of questions answered correctly (5 percent tolerance for transcribed values, 2 percent for derived ratios)
  - Passes when: Score of 0.5 or higher (5 or more of 10 correct)
- **Board-pack narrative**
  - What the agent does: Writes a five-section board report (executive summary, revenue drivers, segment performance, risk factors, and outlook; 80 to 400 words each) for one company from a supplied brief
  - Data and tools: Ground-truth briefs and a scoring rubric; no tools; up to 3,000 output tokens; one model call
  - Score: Mean across sections of keyword credit x length credit
  - Passes when: Score of 0.5 or higher

Preparing the data

- Synthetic financial database, moderate tier: 500 companies, each with one synthetic 10-K filing, a 40 percent discrepancy rate, a 5 percent discrepancy threshold, and random seed 42, generated by the framework's corpus generator
- Filing Q&A retrieval library: 270,806 passages, combining 270,716 passages from 949 real SEC EDGAR filings (Forms 10-K, 20-F, and 40-F; U.S. government public-domain data) with 90 synthetic answer-bearing passages from 10 synthetic filings; the framework splits text into 2,000-character passages with a 200-character overlap
- Filing Q&A answer key: 107 questions about 10 synthetic filings (11 question templates per filing, minus 3 questions that would have revealed their own answer). Each task asks 10 of them
- Board pack answer key: 499 companies (one of the 500 lacked the data for a brief after generation), five sections each

Setting the run parameters

- Difficulty tier: Moderate
- Maximum output tokens per model request: 6,144
- Temperature: 0
- Task pool: 20 task instances per task type
- Filing Q&A retrieval: On, three passages per question, at most 150 passages considered, real-filing distractors on
- Per-task timeout: 600 seconds

How we tested

We ran the agent framework, the model server, and the database as Docker containers on the one server under test. The model server ran on socket 0, and the agent runner, control panel, and database ran on socket 1.

Installing the operating system

1. Install Ubuntu Server 26.04.1 LTS with OpenSSH enabled on the Dell BOSS-N1 boot volume, accepting the default storage layout (LVM with an ext4 root file system).
2. Log into the server, and update the installed packages:

```
sudo apt update
sudo apt -y upgrade
```

3. Reboot the server.
4. Extend the root logical volume over the rest of the boot volume. The installer's default layout leaves most of the volume group unallocated:

```
sudo lvextend -r -l +100%FREE /dev/ubuntu-vg/ubuntu-lv
```

Creating the data volume

We created the data volume on four disks with a script. The steps below are the commands it ran.

1. Clear any partition tables from the four data NVMe drives:

```
sudo wipefs -a /dev/nvme1n1 /dev/nvme2n1 /dev/nvme3n1 /dev/nvme4n1
```

2. Create a RAID 10 array from the four data NVMe drives:

```
sudo mdadm --create /dev/md0 --run --level=10 --layout=n2 --chunk=512 --raid-devices=4 /dev/nvme1n1
/dev/nvme2n1 /dev/nvme3n1 /dev/nvme4n1
```

3. Save the array configuration so it assembles at boot:

```
sudo mdadm --detail --scan | grep /dev/md0 | sudo tee -a /etc/mdadm/mdadm.conf
sudo update-initramfs -u
```

4. Create an XFS file system on the array, labeled data, and create the mount point:

```
sudo mkfs.xfs -f -L data /dev/md0
sudo mkdir -p /data
```

5. Add the volume to /etc/fstab by label, with options that let the server boot if the array is missing or degraded, and mount it:

```
echo 'LABEL=data   /data   xfs defaults,noatime,nofail,x-systemd.device-timeout=30s 0 2' | sudo
tee -a /etc/fstab
sudo mount -a
```

We used the same script to create the 14TB /models volume on the hardware RAID virtual disk (GPT label, one XFS partition labeled models, mounted by label with the same options). That volume held model archives.

Installing Docker and the framework prerequisites

1. Install Docker Engine and the Docker Compose plugin from Docker's official Ubuntu repository. We used Docker Engine 29.8.2 and Docker Compose v5.6.0.
2. To keep container images and volumes on the data volume, set Docker's data root to a folder on that volume. Create /etc/docker/daemon.json with the following content:

```
  {
  "data-root": "/data/docker"
}
```

3. Docker Engine 29 stores image layers through containerd, which has its own root directory. Set root = "/data/containerd" in /etc/containerd/config.toml, then restart both services:

```
  sudo systemctl restart containerd docker
```

Planning CPU placement

We ran the model server on socket 0 and everything else on socket 1, so the agents' data work never competed with inference.

1. List each logical CPU's socket and physical core:

```
 lscpu -e=CPU,SOCKET,CORE
```

2. For the model server, choose one logical CPU on each physical core of socket 0, and leave the second Hyper-Threading thread of each core unused. On our server, CPU numbering alternates between sockets: socket 0 owns the even CPU IDs, and a core's two threads are not always 128 apart. We therefore chose the CPUs from this listing rather than from a numeric range. We used the 64 even CPU IDs 0 through 126.
3. Give the agent harness and the database all logical CPUs of socket 1. On our server, those were the odd CPU IDs 1 through 255.

Deploying the model server

1. Download the model from Hugging Face. Intel publishes it in OpenVINO format with INT4 weights, and no access token is needed:

```
 hf download OpenVINO/Qwen3-8B-int4-ov --local-dir /data/models/qwen3-8b-int4-ov
```

2. Turn off Qwen3's thinking mode by default. In the model folder's chat_template.jinja, add a first line that sets enable_thinking to false when the request does not set it. Without this, Qwen3 writes a reasoning section before every answer.
3. Start OpenVINO Model Server from Intel's image, pinned by digest, on the socket 0 CPUs you chose:

```
docker run -d --name ovms --restart unless-stopped \
  --cpuset-cpus <socket 0 CPU list> -p 8000:8000 \
  -v /data/models/qwen3-8b-int4-ov:/model \
  openvino/model_server@sha256:148e3fe802261ed0b5b634de3bbccdefe7176a218ec34a88e382b40704c34a1c \
  --rest_port 8000 --model_path /model --model_name OpenVINO/Qwen3-8B-int4-ov \
  --task text_generation --target_device CPU --cache_size 80 --max_num_seqs 64 \
  --enable_prefix_caching true --dynamic_split_fuse false \
  --max_num_batched_tokens 32768 --metrics_enable
```

Table 7 explains each setting. During our testing, the server's command line was:

```
/ovms/bin/ovms --rest_port 8000 --model_path /tmp/ovms-serve --model_name OpenVINO/Qwen3-8B-int4-ov
--task text_generation --target_device CPU --cache_size 80 --max_num_seqs 64 --enable_prefix_caching
true --dynamic_split_fuse false --max_num_batched_tokens 32768 --metrics_enable
```

Deploying the database and agent harness

1. Run PostgreSQL 16.15 (postgres:16-alpine image, pinned by digest sha256:721873c34ceb9f8d8fc265984940dc982404c105f19ad51be9fdc5970a6080ea) in a container on the socket 1 CPUs.
2. We drove the workload with the PT agentic benchmark framework, an internal PT tool that is not publicly available. It ran in its own container on the socket 1 CPUs and:
   - generates the synthetic financial data and loads it into PostgreSQL;
   - runs LangGraph agents that send chat requests to OVMS through its OpenAI-compatible endpoint (/v3/chat/completions);
   - keeps a fixed number of agents busy for each measurement;
   - grades every answer and records each task's timing.
3. We limited the agent processes' math libraries to one thread each (OMP_NUM_THREADS, OPENBLAS_NUM_THREADS, and MKL_NUM_THREADS set to 1), so parallelism came from running many tasks at once.
   a. To build a comparable test, use an agent framework that calls an OpenAI-compatible endpoint, pin it and its database to the socket not used for inference, and hold a constant number of concurrent agents for each measurement window.

Preparing the financial data and retrieval library

We used the PT agentic benchmark framework's data generator to create the moderate-tier synthetic financial data: 500 companies, each with one synthetic 10-K filing, plus their transactions, positions, market data, and regulatory ratio inputs (a 40 percent discrepancy rate, a 5 percent discrepancy threshold, and random seed 42). The framework loaded the data into PostgreSQL.

For the Filing Q&A task, the framework downloaded public annual-report filings (Forms 10-K, 20-F, and 40-F) for about 1,000 companies from the SEC's EDGAR system, 949 filings in total, split them into 2,000-character passages with a 200-character overlap, and built a search index over them. It then added 90 passages from 10 synthetic filings that contain the answers, for a library of 270,806 passages.

To build a comparable test, use any set of public SEC filings split into passages of similar size, with a known set of answer-bearing documents hidden among them.

Verifying the deployment

1. Confirm that the model server reports the model as available:

```
curl -s localhost:8000/v1/config | grep AVAILABLE
```

2. Confirm the server command line and the CPU pinning of each container:

```
docker top ovms -eo pid,args
docker inspect ovms <database container> <agent harness container> --format '{{.Name}}
{{.HostConfig.CpusetCpus}}'
```

3. Confirm that PostgreSQL accepts connections:

```
docker exec <database container> pg_isready
```

4. Send one chat request to the model server, and confirm that the reply contains only the answer, with no thinking section (no <think> tag):

```
curl -s localhost:8000/v3/chat/completions -H 'Content-Type: application/json' -d '{"model":
"OpenVINO/Qwen3-8B-int4-ov", "messages": [{"role": "user", "content": "What is 17 times 23? Answer
with the number only."}], "max_tokens": 64, "temperature": 0}'
```

   a. The reply's message content should be 391.

Running the measurement sweep

Each measurement is a run at one concurrency level, where the concurrency is the number of agents the framework keeps busy at once. In each run, every agent starts a new task as soon as its previous task finishes, so the load stays constant for the whole measurement window.

We ran the sweep from a saved configuration in the framework's control panel, so every run used the same settings: moderate tier, a 420-second measurement window, a 15-second warm-up, a 600-second prewarm on the first run, a 600-second per-task timeout, a maximum of 6,144 output tokens per model request, and temperature 0.

1. The sweep first ran 28, 31, and 34 concurrent agents, three times each, in interleaved order (28, 31, 34, 28, 31, 34, 28, 31, 34). This placed the 30-second p95 target between 28 and 31 agents.
2. The sweep then ran 29 and 30 agents three times each on its own, to find the exact limit. Other configurations narrow to different values.
3. We ran the full sweep three times to check consistency.

To build a comparable test, hold each concurrency level for a fixed measurement window after a short warm-up, run each level more than once in interleaved order, and narrow the search between the highest passing and the lowest failing level.

Timing each cell

- Prewarm: The first cell of the sweep runs 600 seconds of real load before its measurement window to warm the server's prompt cache. The framework discards those tasks.
- Warm-up: Every later cell runs 15 seconds of discarded load before its window.
- Measurement window: This is 420 seconds at the cell's concurrency. In every cell of all three sweeps, the window lasted exactly 420 seconds.
- Drain under sustained load: When the window closes, the agents keep starting replacement tasks until every task started in the window has finished, so the slowest tasks finish under full load rather than on an emptying server. The replacement tasks never count toward steady-state throughput or task times; only all-completed throughput includes them.
- Task timeout: This is 600 seconds, the only bound on the latency of a single task. A task that reaches the timeout stays in the results at the bound, so a truncated task raises the p95 latency rather than disappearing.

Determining M

M is the maximum number of concurrent agents for which the p95 task latency stays at or under the 30-second service level. We chose 30 seconds because a person reviewing a financial answer would likely expect a response in under 30 seconds; it is also the framework's service-level target for this suite.

- p95 for a cell: This is the 95th percentile of the latency of every task the agents started during the 420-second window, however long the task ran. Tasks from the warm-up, the prewarm, and the drain replacements do not count.
- Broad phase: The sweep runs 28, 31, and 34 agents, three times each, in interleaved order, to bracket the service level.
- Refinement phase: The sweep tests levels between the highest passing (28 in our case) and the lowest failing level (31 in our case), three times each, until the two are one agent apart. The refinement phase ran with 29 and 30 agents.
- We ran each concurrency level three times. For each level, we selected the run with the median steady-state tasks per minute (the earlier run when two tied) and took every value for that level from it. M is the highest level whose selected run had a p95 task time of 30 seconds or less.

Calculating throughput

- Steady-state throughput (the throughput we report): This is the tasks that completed inside the 420-second window, divided by 7 minutes. Because the window excludes the warm-up and the drain, this figure does not depend on how long a run takes to start or finish.
- All-completed throughput: This is every task that finished during the run (including the warm-up, the prewarm in the sweep's first run, and the replacement tasks during the drain) divided by the run's total elapsed time. It reads lower than steady-state throughput because the warm-up and drain add time when fewer tasks are finishing.

Configuration tunings we evaluated

Before the measurement ran, we tested changes to the model server and the model at a fixed load of 40 concurrent agents. These changes included prompt splitting (dynamic split fuse), an 8-bit KV cache, dynamic quantization, and giving OVMS the Hyper-Threading sibling threads of its cores. We kept a change only if it raised throughput by at least 3 percent without worsening p95 latency. None met that threshold, so we used the configuration in Table 7.

This project was commissioned by Dell Technologies.

Read the report ▶

Primary contributors

Tech: Corey B.

Writing: Laura W.

Design: Laura K.

PM: Greg C.

How we created this report

A PT team, which includes the contributors we've listed and others, created this report and performed the technical work behind it. In addition to our use of AI in testing, we used AI to help create charts, flow content into layout, and draft and proofread sections of this report.

Principled Technologies®

Facts matter.®

Principled Technologies is a registered trademark of Principled Technologies, Inc. All other product names are the trademarks of their respective owners. For additional information, review the science behind this report.

DISCLAIMER OF WARRANTIES; LIMITATION OF LIABILITY:

Principled Technologies, Inc. has made reasonable efforts to ensure the accuracy and validity of its testing, however, Principled Technologies, Inc. specifically disclaims any warranty, expressed or implied, relating to the test results and analysis, their accuracy, completeness or quality, including any implied warranty of fitness for any particular purpose. All persons or entities relying on the results of any testing do so at their own risk, and agree that Principled Technologies, Inc., its employees and its subcontractors shall have no liability whatsoever from any claim of loss or damage on account of any alleged error or defect in any testing procedure or result.

In no event shall Principled Technologies, Inc. be liable for indirect, special, incidental, or consequential damages in connection with its testing, even if advised of the possibility of such damages. In no event shall Principled Technologies, Inc.'s liability, including for direct damages, exceed the amounts paid in connection with Principled Technologies, Inc.'s testing. Customer's sole and exclusive remedies are as set forth herein.