EXASERVE: EXTREME-SCALE INFERENCE

Framework Reference Card

Scaling LLM Inference on HPC — One OpenAI-Compatible Endpoint from N Compute Nodes • https://github.com/globus-labs/exaserve

ExaServe is a framework for scaling LLM inference across the compute nodes of an HPC system. From one YAML file and one command, it turns a batch-scheduler allocation (e.g. a PBS job) into a single OpenAI-compatible inference service: it launches a Ray cluster over the allocation, stages model weights to node-local storage with an MPI broadcast, deploys one inference replica per GPU tile as Ray Serve applications — a single EngineWorker host over a pluggable engine backend (vLLM or SGLang) — and optionally fronts them with a pluggable head-node proxy such as HAProxy.

1 · Why ExaServe?

Existing LLM serving stacks are designed for cloud environments; HPC systems bring batch-queue scheduling, MPI-only launch paths, Lustre metadata costs, exotic accelerators, and scale cliffs that only appear past a hundred nodes. ExaServe packages the engineering needed to cross that gap:

Turnkey N-node serving

One YAML file plus one command turns a batch-scheduler allocation into an OpenAI-compatible service; clients see the standard API and need no HPC knowledge.

Validated at scale

Deployments measured to 256 nodes / 3,072 replicas behind a production HAProxy front end: 27.1k non-streaming QPS with Llama-3-8B (one replica per tile) — 96% weak-scaling efficiency at 256 nodes (27.1k of 28.2k offered, 0% errors).

Frontier-model ready

Multi-node pipeline parallelism serves Llama-3.1-405B (TP8 × PP2, two nodes per replica) with shard-aware weight staging, demonstrated from 2 to 128 replicas (4 to 256 nodes) at 67% weak-scaling efficiency (streaming).

Measurement built in

A declarative benchmark harness generates traces and scheduler jobs, replays load through a Go client, and scores per-request Time-To-First-Token(TTFT)/Time-Between-Token(TBT) against latency SLOs.

2 · What's in a deployment?

A deployment is a small stack of cooperating pieces, all driven from a single configuration file:

Component	Description
Launcher	exaserve-serve-submit (batch, from a login node) or exaserve-launch-cluster (interactive) bring up the whole stack on a scheduler allocation
MPI weight staging	One-source-many-sinks MPI broadcast from the shared file system (e.g., Lustre) to node-local storage; shard-aware per pipeline stage for very large models
Inference replicas	vLLM or SGLang replicas as independent Ray Serve applications — one per GPU tile (12 per Aurora node) for single-tile models, or spanning multiple tiles and nodes via tensor/pipeline parallelism for larger models
Head-node proxy	HAProxy, LiteLLM, and others — a single client-facing endpoint with load balancing, plus auth and rate limits with LiteLLM
Site patches	Site-specific patches to Ray/vLLM required for scaling on HPC systems
Benchmark harness	A simple evaluation harness is included to evaluate performance. Declarative specs → traces + scheduler jobs → replay → SLO scoring

3 · Installation

Install steps are site-specific — they depend on how Ray, the inference engine, MPI, and the accelerator toolchain are provided by the host. The recipe below targets ALCF Aurora; recipes for other sites will be added as they are supported.

ALCF Aurora

The package installs into the Python provided by Aurora's frameworks module, which already includes Ray, vLLM, MPI, and the oneAPI toolchain; pip adds only the ExaServe package itself.

```
module load frameworks
git clone https://github.com/globus-labs/exaserve && cd exaserve
python3 -m pip install --user .

# console scripts land in a frameworks-versioned bin dir; add it to PATH:
export PATH="$(python3 -c 'import sysconfig; print(sysconfig.get_path("scripts", "posix_user"))'):$PATH"
which exaserve-serve-submit   # verify
```

One-time additional steps, each needed only for the feature that uses it: build HAProxy with scripts/build_haproxy.sh (proxied deployments); create a separate LiteLLM venv (its dependencies conflict with Ray/vLLM); build the benchmark load generator with module load go && bash eval/go_client/build.sh.

4 · Example

ALCF Aurora

One YAML file describes a deployment — the Ray cluster, the model deployment, and the client-facing proxy. Submit it from a login node, then query the printed URL from anywhere that can reach the head node:

```
exaserve-serve-submit my_config.yaml --project-account YOUR_PROJECT --wait
# 8470123.aurora-pbs-0001...          <- PBS job id
# http://x4310c1s0b0n0:4001           <- service URL

curl -sS -X POST "http://x4310c1s0b0n0:4001/v1/chat/completions" \
     -H 'Content-Type: application/json' \
     -d '{"model":"meta-llama/Meta-Llama-3-8B-Instruct",
          "messages":[{"role":"user","content":"hello"}],"max_tokens":8}'
# {"id":"chatcmpl-...","choices":[{"message":{"content":"Hi!"...}}],...}

qdel 8470123                          # tear down
```

The configuration it references: Llama-3-8B on two nodes (24 replicas) behind HAProxy. model_storage_path (your model directory on the shared file system) and local_stage_path (node-local staging) are typically the only fields that vary per user.

```
# my_config.yaml
ray_cluster_config:
  head_ip: ""              # filled in at runtime
  port: 6379
  node_cpus: 64
model_deployment_config:
  num_nodes: 2
  model_storage_path: /lus/flare/projects/<PROJECT>/<user>/models
  local_stage_path: /tmp/hf_home
  num_gpus_per_node: 12
  model_configs:
    - model_id: meta-llama/Meta-Llama-3-8B-Instruct
      tensor_parallel_size: 1
      pipeline_parallel_size: 1
      max_model_len: 4096
      size: 8               # billions of params; drives replica planning
      gpu_memory_utilization: 0.90
      enforce_eager: true
      max_num_seqs: 64
proxy_config:
  type: haproxy             # haproxy | litellm | none
  port: 4001                # client-facing port
  backend_port: 8000        # Ray Serve HTTP port on each node
```

Learn more: the README covers interactive launches inside qsub -I, custom PBS integration via --dry-run, and every configuration field (examples/config.reference.yaml is fully annotated).

5 · Serving agents and OpenAI-compatible clients

The deployment exposes the standard OpenAI API, so any compatible client works unchanged — LangChain's ChatOpenAI, Academy LLM agents, litellm, or the openai SDK — by pointing the usual environment variables at the service URL. This makes the framework the self-hosted, on-machine complement to a hosted gateway: the same agent code runs against either.

```
export OPENAI_BASE_URL=http://<head_node>:4001/v1
export OPENAI_API_KEY=EMPTY
```

There is no authentication by default; use proxy_config.type: litellm to add API keys, rate limiting, and usage tracking.

6 · Examples and Resources

Runnable configurations and benchmark specifications are in the GitHub repository; code is linked rather than copied here.

Resource	Location
Source code	https://github.com/globus-labs/exaserve
Documentation	README.md — install, configuration, console-script reference
Machine-actionable card	doc/exaserve.md — markdown version of this card for coding agents
Deployment templates	examples/ — HAProxy, LiteLLM, and a fully annotated reference config
Benchmark specs	eval/specs/refcard/ — the smoke, weak-scaling, and 405B pipeline-parallel specs referenced in this card
Scaling analyses	findings/ and doc/KNOWN_ISSUES.md — root-cause write-ups, scale cliffs and fixes