NVIDIA Solutions architect

~~Augmenting LLM using ...~~

Canonical RAG workflow:

RAG vs Fine-tuning:

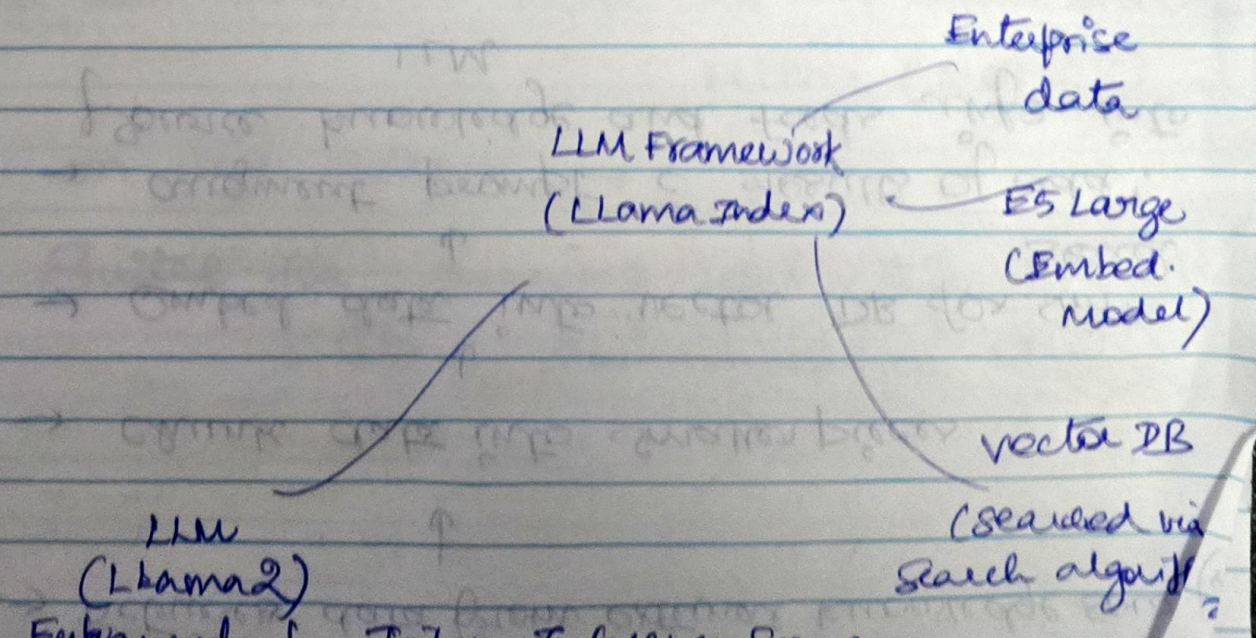
- can be combined to leverage strength.

- FT: changes style/tone of response

add new data - impossible / difficult

RAG = Facebook (2020) → generator + i/p query + chunk

GPT: ~~B~~



NVIDIA Solutions architect

~~Augmenting LLM using ....~~

Canonical RAG workflow:

RAG vs Fine-tuning:

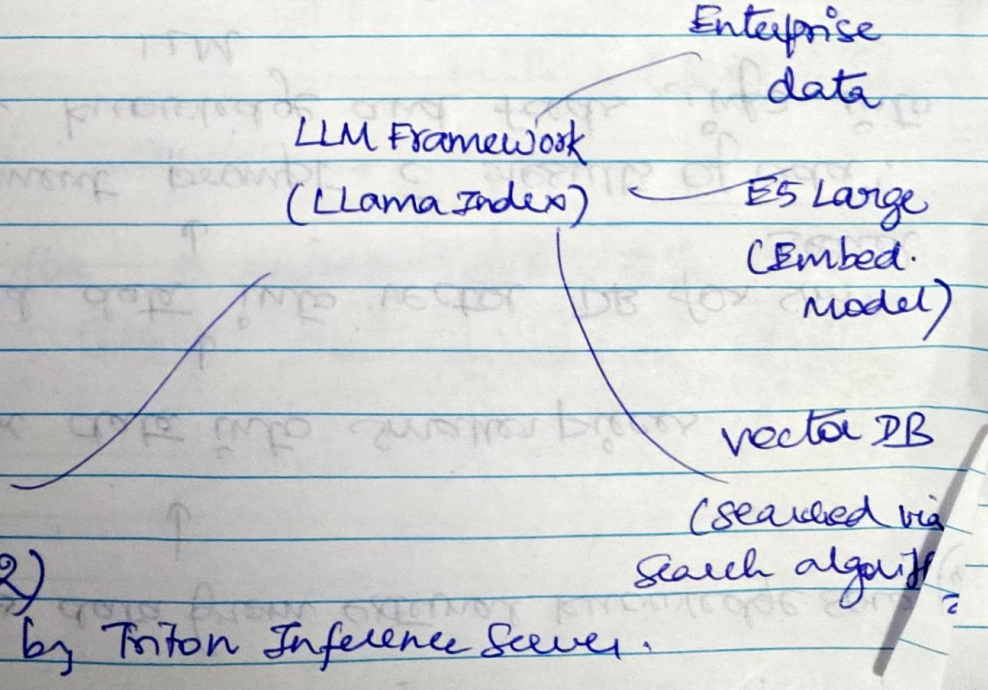
- can be combined to leverage strength

- FT: changes style/tone of response

  add new data - impossible / difficult

RAG = Facebook (2020) → generator + I/P + chunk query

GPT: ~~B~~



CLASSIC RAG

Steps in RAG!

→ Retrieves data from external knowledge source

↓

→ Chunk data into smaller pieces

↓

→ Embed data into vector DB for sim. search

↓

→ augment prompt ~~c~~ results of add". source knowledge and feeds info into LLM

Agent - assited RAG :-

CoT based RAG : ↻ with more toolcalls

Decompose complex query into simple ones

RAG ~~c~~ guardrails :-

Speeds up decision making process (COT) when using agents.

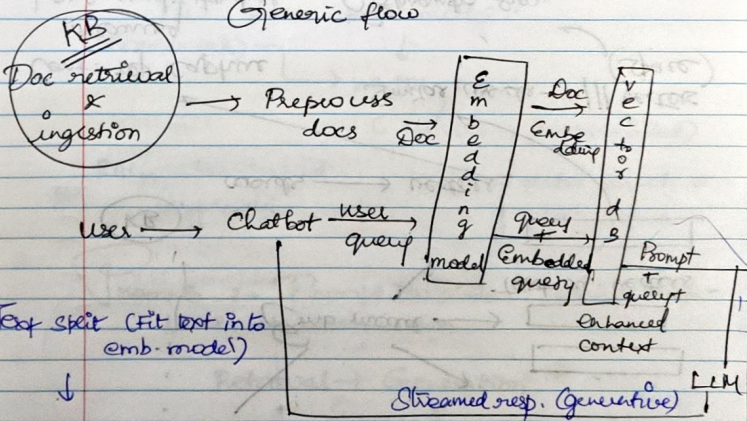
→ Outputs align ~~c~~ high quality

guardrails :- a)

b) safety guardrails - filter unwanted content

c) security "

Generic flow



Text split (Fit text into emb. model)

↓

data → High dim vectors → vector DB → rapid search.

Milvus (RAPIDS)

Trained on vast datasets →

Suited for querying ext. KB

Q&A, Summarization, Chatbot

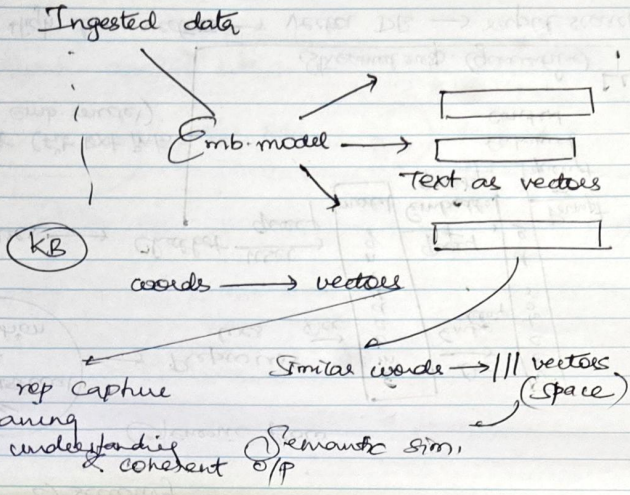
Ingestion

Query & passages → Retrieve Search
- DB
- doc
- HTML
- .md

→ Process Raw txt
- clean
- chunk
- remove dups

→ tokenize chunks ↓ tokens

Embedding generation



In-memory / vector store

Vector DB → high dim vec.

FW (Framework) → Emb. model → vector DB

(Milvus → purpose filled)

RAG → Pre-trained Lang. model

Redis → vector search is add^n feature.

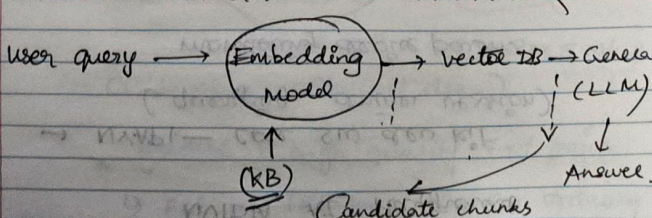
Prompts :- Prompt Template - useful high-level ~~as~~ abstraction

Retrieval + Generation

prompt + context

NVDA + RAG : Soln. local inference :

Latest Tesla + GTX drivers, Llama ?



NVIDIA AI Playground

→ NVAPI — Core SW dev kit
(querying driver version)
monitoring device params

★ Interface to Llama 2 API.

'Langchain — General framework (matured)

Llama Index — III to langchain's Index/query tasks are good!

Foundation	Chain	Agents
— Model wrapper — Prompt template — Indices / vector store — memory — docu loader / text splitters	— combine multiple comps to solve an issue (chat w pdf, analyze data)	— tool-calls to complete task

Mature agents and mem. structures

LlamaIndex

Data connectors → docs data Indices → Query

(docs from directory) | Structure data for diff use cases | Interface [Feed in i/p prompt & obtain knowledge aug. o/p]

Reddit / SB
↓
Consistency based filter (1.3B) — Embedding model (E5 large)
↓
CC-pairs (2TOM) — TLM (Transaction Level Modeling)
More eff on GPU — Commercial license!

technique → Contrastive loss

Convert txt to array of FP #s

Query emb — Query emb
↑ Pool — Pool
↑ Encoder — Encoder
text1 — text2
Shared wt. (SIAMESE N/W)

CC-pairs (Clausal clean)

Diverse & high quality txt pairs
↳ Reliable txt pair used for training

Vector DB: Milvus

* Query vect ⟷ vect in collection
DISTANCE

Why milvus? RAFT accelerator lib supported
↳
Heterogeneous Comp → C GPU support, faster and better perf.
Scaling to more doc → better
vector DB integrated C
Chroma, FAISS, MongoDB.

RAPIDS RAFT: Accelerate ML algo on GPU
for vector search
Stores index & massive vol of emb vectors
C off-Triton LLM server (Inf) Server

trained LLM: Meta's Llama?
70B params
[perform better than OS models]
SFT: → RLHF (Rejection Sampling and PPO)

Why Triton simplifies Inference?

Any Framework
Any Inf type
(Real type Batch)
GPU+CPU
high tput + low latency

Model Inf.
CPU/GPU/
Cloud/edge
[Designed for MLOps/WebOps]
kHbernetes, AWS, GCP AI,

RAG:
- more accurate due to RT data
- Less vulnerable to hallucination
- Enhanced transparency
- Leverage Internal Knowledge source

Sensitive data on-prem

Cons of RAG:
- Tech prof.
- Data freshness/retrieval strategy
- vect search — slow!
- Inadvertent release of sensitive info.

Model FT C NeMo. [guardrails for style]
RAFT
Speeds up!
NeMo guardrails

Vector DB: Milvus

NeMo: NVidia Maximum Optimized

Framework | Model | API

a) Neural module – Speech recognition, language understanding & resp gen

b) Experiment mgt – training & eval of NeMo models – perf / hp. of NeMo models

c) Checkpting: Save & load model states during training

Features:-

Mixed precision training (Comb of FP16 & 32 train – ↓ mem usage & perf

Parallelism: train model on multi-GPU – ↓ reduces training time & ↑↑↑ Scalability

Optimization: Attention optimization, activation recomputation and comm overlap – to ↑ model perf and efficiency

Use of NeMo:-

Chatbot | virtual asst | speech recog

VLM → (a) NeVA (LLaVa) – text out of img

(b) LLaVa-next → ↑↑↑ LLaVa

(c) Llama 3.2 vision models: generate text based on imgs

Speech models (a) Diffusion models (high quality speech)

(b) Flux → Speech AI model that generates speech in real-time

Emb models: SBERT, Llama Emb

Tools: NeMo Run → tool to run NeMo models & exp.

PT models – Mixed precision training

NeMo agent toolkit

* nat run --config_file x --input ....
  [Test workflow]

* nat serve --config_file x --host ... 127.0.0.1.
  [Deploy]

How to write config file?

% writefile config.yml

```
llms:
  climate_llm:
    _type: nim
    model_name: meta/...
    base_url:
    api_key:
    temperature:
    max_tokens:

workflow:
  _type: chat_completion (react_agent)
  llm_name: climate_llm
  system_prompt: |
```

[No quotes]

Above agent answers generic qns. on climate via LLM data!

Next step:- ★ Tag Python tools into NAT- compatible tools

★ Register multiple tools & ReAct agent workflow

★ Agent decision making & tools

```
@ register_function(config_type = ...)
async def cal_stats(...):
```

All tools begin & conf. class

```
class Calc_stats_config (FunctionBaseConfig,
                         name = 'calc_stats'):
    pass

class CalcStatsInput (BaseModel):
    Country: str = Field (default = '',
                          discription = '...')

@ register_function (config_type = CalcStatsConfig)
```

.toml file :- [project], [project.entry-points.nat]

→ Observability in NAT :-

→ understand what's happening in your workflow

* → Phoenix → Server by Arise

General:
telemetry:
tracing:
phoenix:
type :
endpoint : (server)
project :

→ Multi-agent integration adding math!

result

→ nat eval --config_file x →

NVIDIA H100: Fast calculator

② Data center GPU on Hopper arch. (Better than A100)

Tensor cores: Super-smart math whiz to do math problems
- ⓐ recognizes pic and videos
- ⓑ understand what ppl say
- ⓒ play games that need math

SXM5

PCIe

NVLink – Superfast internet connection to talk to other comps and share info

A lot of memory; → ⓐ train AI models
- ⓑ Scientific sims
- ⓒ play games

Tensor core: 4th gen – supports FP8 and BF16 dtypes
(Improved perf for AI/HPC workloads)

Hopper Streaming Multiprocessor:-
Building block of H100 GPU.
- 128 FP32 core, 64 INT32 and 4 Tensor core

NVLink: High Speed interconnect to enable fast data transfer b/w GPU and other comps

Specs: ⓐ Peak FP64 tensor core perf upto 160 TFlops (SXM5) and 48 TFlops (PCIe)
- ⓑ FP32: upto 160 TFlops and 48 TFlops
- ⓒ FP16: 1120 " 96 "
- ⓓ INT8: 12000 TOPS and 3200 TOPS

Memory and BW:
- HBM2e memory – 80 GB & BW of 2K GBPS (SXM5) and 1155 GBPS (PCIe)
- L2 cache – 50 MB (SXM5) and 40 MB (PCIe)

A100 – Ampere arch.
H100 – Hopper

Streaming Multiprocessor: Building block of GPU. Multiprocessor that executes multiple threads concurrently (using a combination of units, register and memory)

Hopper arch (SM) features:
- 128 FP — Scientific sims, DL and compute heavy work
- 64 INT — data proc, encryption and compression
- 4 Tensor core: Specialized for matrix mult. and acc. → DL/AI workloads

TFLOPS: Teraflops – measure of GPUs peak FP perf. # FP ops/sec

TOPS: Teraops → Peak INT perf (# Int op/sec)

~~Design of Dual Brain Teacher Agent~~

HBM2e → High BW memory 2e
↳ Memory type in H100.

High BW and Low latency

NVLink could be fastened data transfer b/w GPU and other parts (say CPU)
– reduces overall training time

GRID K520 GPU	H100
– 1.3 TFLOPS (peak single precision FP perf).	– 160 TFlops (SXM5) & 48 TFlops (PCIe)
– 0.65 TFLOPS (double precision FP perf)	– 130 TFLOPS (SXM5) & 24 TFLOPS
– Mem BW 208GBPS	– 2000GBPS (SXM) & 1155 GBPS (PCIe)
– CUDA cores : 1536	CUDA cores: 14592 (SXM5) and 5888 (PCIe)

123 × Faster than GRID K520

Roadmap of GPU Arch:

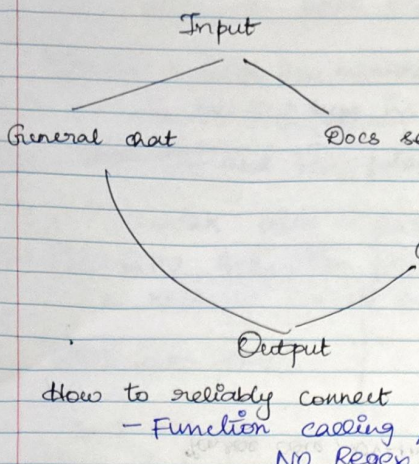
Fermi → Kepler → Maxwell → Pascal → Volta → Turing → Ampere → Hopper

1080 Ti = 11.3 TFlops (FP32) [Pascal]
↳ 3.5K CUDA core
484 GBPS Mem B/W

0.3 TFlops (FP64)

(vs) 160 TFlops (H100)
130 TFlops (SXM5)

Tensor core, NVLink & PCIe Gens



How to reliably connect LLM to DB?
- Function calling / structured o/p
NO Regex

```
class RouteQuery (Base Model):
    datasource : Literal['vectorstore',
                         'general chat']
```

LCEL / LangChain Expression Language

grader_chain = grade_prompt | llm.with_structured_output (GradeDocument)

[O/p of Left → I/p of Right)

Ans: structured responses

MCP Server — Defines connecting interface to tools/data

Old — write py fn. for each agent
New — Server exposing tools — agent connects tools like USB device

LangGraph — Define state and Flow!

Chain : input → o/p (Stateless)

- they need memory of what happend in prev. steps

```
Class GraphState (TypedDict):
    question : str   # OG i/p
    generation : str  # final ans
    context : str   # raw text in DOCS
```

Start → Node 1 → Gen

{"qn": "..."} → Router → {"output": ?}

How did you build a router?
→ used pydantic model RouteQuery & structured output to ensure type safety
→ How did you manage state? LangGraph

Why can't I shove NVIDIA documentation into prompt?
- Cost
- Accuracy
- Latency ⊖ .

→ Web Scraping vs cleaning :

WebBase Loader powered by bs4
(Raw HTML – full of noise – HTML tags)

RAG acc → Data cleaning - ✗
Strips tags and leaves just text content

Chunking :- RecursiveTxt CharacterSplitter
↳ Chunksize : 1000
Chunkoverlap: 100
[Preserves semantic coherence]
↳ Splits by paragraphs first
↳ then sentences
↳ if not, words

Embeddings : NVIDIA nv-embedqa-e5-v5

Trained specifically for QA pairs
hence performs better on technical queries than OAI embeddings

Design overview:

Tutor for SA - NVIDIA role
to know more abt products of NVIDIA
- that should teach me and help co-create stuff ☆ resume
- test me
- contain study matl. to official NVDA docs

Multiagent system using LangGraph
(a) Router – Llama 3.1-70B to intent classify
(b) Tutor – searches SDT Doc pgs and stores in RAG
Retrieval : (used E5 embedding to search) + FAISS
Reasoning : Sequential thinking loop (Analyze, Plan and Teach) – CoT before gen. tokens
(c) Contextual Brain / Generic Tutor :
Ingest PDF to vector index (Target skills dynamic ah)

why langgraph:
- state machine (NOT DAG!)

Retrieval + thinking in ||el

[Streamlit's model visualizes it for debug]

NVIDIA AI endpoints – connects to NIM
[NVDA Inference Microservices]

ChatNVIDIA → points to build.nvidia.com
library

cluster of NIM containers
running on HIDOS

Tensor RT – SDK to allow developers to optimize for spec GPU arch, deploy on GPU for efficient inference

model opt → reduces # params & comp

Quantize → reduce precision of model wt

pruning → remove unnecessary wts. & conn

Know. Distill. → transfer know from → model for edge

Agent

Memory → agent's train of thought [semantic sim, recency, importance, appn. spec metrics)

user req. — Agent core — Planning module

Tools

Core logic and behavioral char.
[Persona of agent]

ReAct, Reflexion, CoT, GoT: Reflection / critic
[Evidence based prompting framework]

Milvus DB → better on perf. and scale on GPU arch.

→ expose docker to all GPUs
→ expose API key
→ NIM downloads + store model wt.
→ Run as local user
to confirm service is ready---

curl →
curl localhost:8000/v1/health/ready

Send a req in CLI:

Request:

```
curl -X 'POST' 'http://0.0.0.0:8000/v1/chat/completions
  -H 'accept: application/json'
  -H 'Content-Type: application/json'
  -d '{
        'model':
        'messages': [{'role': ' ', 'content': ' '}]
        'max_tokens':
      }'
```

NeMO agent toolkit

- Framework agnostic
- Profiling / observability
- Eval system

```
functions:
    current_dt:
      _type:
    getting_started:
      _type:
      prefix
llms:
    nim_llm:
      _type: (nim)
      model_name:
      temp:
workflow:
    _type:
    llm_name:
    tool_names:
```

@ register_function → decorator to inform a toolkit that a fn. should be accessible by name when referenced.

↳ specify config_type

Params of decorated fn: FunctionBaseConfig, builder
↳ dynamically query + get other workflow fn.

NAT relies on yield ~~return~~

```
-nat run --config_file
         --input " "
```

```
async with load_workflow(config_file) as workflow:
    async def single_call(input_str):
        async with workflow.run( ) as runner:
            return await runner.result( )
```

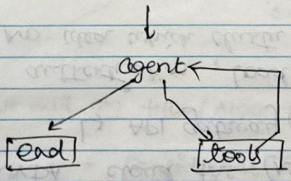
yield :-

```
async for callable in get_callable_for_workfl
                                    ( ):
```

```
langchain_nvidia_ai_endpoints import
                                    ChatNVIDIA
llm   = ChatNVIDIA(llm='meta/...')
tools = [TavilySearch(max_results=1)]
                        tools
llm_with_tools = llm.bind_tools(tools)

llm_with_tools.invoke(      ).tool_calls

        [t.name: t for t in tools]

tool_call = ai_response.tool_calls[0]
          - [t.name: t for t in tools]

'agent'

add_edge('__start__', 'agent')
add_conditional_edges('agent'
                      should_continue)

workflow.add_edge('tools', 'agent')
app = workflow.compile()
```

```
def should_continue(state):
    messages = state['messages']
    last_message = messages[-1]
    if last_message.tool_calls:
        return 'tools'
    return '__end__'
```

Start




```
for chunk in app.stream(
    {'messages': [('human': 'what's the
                    weather in SF?')]},
    stream_mode='values'):
        chunk['messages'][-1].pretty_print()

chunk['messages'][-1].pretty_print()
```

NIM handles model ~~selection~~, optimization / backend selection and scaling automatically

Basic docker commands

* Optimizes LLM deployment

- OpenAI compatible REST APIs
- Deploy models from HF Hub
- Deploy models from local storage

Inference backends :

(a) TensorRT-LLM (high perf)
(b) VLLM (versatile deployment)

Container runtime → Docker & NVIDIA container runtime

NIM API

Request lifecycle :

Client → NVDA cloud → Orchestrator (K8) → NIM container

NVDA cloud fns. (NVCF)
↳ API gateway (serverless)

↓ GPU

authentication, load balancing & routing
No idea which cluster you're running

Orchestrator (Kubernetes) → DGX cloud
↳ routed to specific pod on ~~NVIDIA~~ NIM container
Autoscale → spins more pods & GPUs (K8)

- NIM Container - highly optimized server stack

(a) API server
- FastAPI / uvicorn
[OAI JSON req to tensors]

(b) Triton Inference server
(manages queue - 100 reqs in, Triton batches them)
(Dynamic batching)

(c) TensorRT-LLM
↳ Doesn't run raw PyTorch code - runs optimized plan for H100s

↓

Mat mul on H100 / A100 GPUs

Why local deployment?
- Confidential data - medical data
- latency - RTT to web
- cost @ scale → Rent / token ↑↑
Rent of GPU - flat rate ↓↓

ChatNVIDIA : langchain_nvidia_ai_endpoints

integrate.api.nvidia.com → localhost:8000

LIFECYCLE

Request hits NVCF API gateway → routed to K8 pod

↓

Triton Inf server batches req

passes to TensorRT engine where mat_mul happens (Optimized for H100s)

Executes Optimized kernels

ReAct → more LLM calls
→ prompt sensitivity and tuning overhead
→ possible risk of hallucination
→ lack of determinism
→ complex long chain

Tool call: User query → fn. match → tool exec
↓ resp.

→ Needs an LLM that supports tool call

nat run —
serve — server that listens to incoming req.
eval — accuracy

functions: → wikipedia_search:
llms:
workflow:
_type:
max_results:
nim_llm:
_type = nim
model_name:
temperature:
_type:
tool_names [ ]
workflow
_name:

Tensor RT - LLM: not just a runner. It's a compiler.
↳ takes in llama3 and rewrites math to perfectly match H100 GPU.

Standard batching
user A — qA (short) (WAITING)
user B — qB (lengthy)
BLOCKING

(a) Soln :- TRT-LLM evicts user A's req the moment it's done and slots in user C even while user B is generating

Max^m. GPU util - we don't pay idle compute → Dynamic / continuous batching

(b) Paged Attention (KV cache optimization)
— LLM needs to remember prev. Tokens [KV cache]

— Standard mem. alloc. reserves huge blocks of vRAM in case answer gets long
(wastes 20-50% GPU mem)
↳ FRAGMENTATION.

Solution architect

[Pre – sales]

Architect - graph pieces together and bring them into a working soln.
work c engineering to fix bugs

Mission - support broader community to adopt AI technology

SW engg. comp than HW manufacturer

= Multiple triton inference server (customers)
= work c pdt. mgmt to resolve → Engg. calls
= Support proposal dev. (sizing, scaling etc;) prompt engg.

NVIDIA product knowledge:
- NIM agents blueprint (Reference workflow)
- Model served as NIM container
- NeMo guardrails to ensure model doesn't o/p proprietary code to public server
- Deploy on triton inference server running on HGOs to leverage transformer for FP8

How would you explain about NIM to someone who knows Docker?

NIM - prepackaged - TensorRT, Triton, Model cot. compiled for GPU arch.

docker run nvcr.io/nim/llama3

All you need is this - no need to install anything

~~the~~ NeMo Framework (vs) Guardrails :-

Creation phase (training and fine-tuning)

Runtime phase (safety layer b/w user and LLM to enforce topic, safety and security)

MCP protocol :-

- Standardizes AI agents to connect to data sources
- Instead of 50 API connectors, you build one MCP server — solves n^2 integration pbm

Optimize a reasoning agent :

- FP8 (H100s) or INT4 quantization using TensorRT LLM to reduce mem. bw. pressure
- Optimize system prompt to minimize reasoning steps
- Speculative decoding : small model to predict tokens and big model verifies them

RAG system that does real-time updation :

- Real-time vector DB : Milvus / Weaviate

Standard RAG : slow to index

Hybrid search : BM25 + vector search

↓

for most recent logs (KW search)

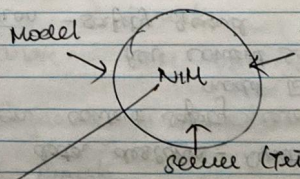
Real-time req. is met w/o waiting for embedding job to finish

- Do we build MCP servers for users ?
- AI Blueprints / custom arch. from scratch

Why NVDA ?

→ NIM, NVLink, H100, TensorRT LLM,

NVDA → OS for AI.



[LLM in a box] → You can run it anywhere!

OpenAI → Runs only on OAI servers.

(a) portability (b) Data privacy (Deploy NIM within company's firewall) → Eg banking) (c) Customization

You can't tell chatgpt to use RPS to save money

- Portability
- Sovereignity
- Customization

NeMo data designer - Synthetic data gen
NeMotron content safety reasoning — guardrail model to handle custom policy for content safety & topic following
Nemotron - safety guard

Reducing KV cache bottleneck:

- grows ∝ prompt length and strains GPU memory
- Limited GPU mem:
  - (a) tradeoff in scalability
  - (b) reduced perf
  - (c) exp. recompute
  - (d) more GPU

NVDA Dynamo : offload KV cache from GPU mem to CPU RAM / SSD / NW storage using low-latency NIXL transfer lib

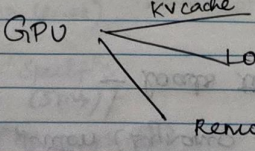
Dynamo's KV Block mgr:

(Router)

- separates mem mgmt from model engines, standardize storage access, simplifies integration for scalability
- offload / reload KV cache @ extremely ↑↑ tput, min. latency and maintain perf

Dynamo ↔ LMCache (flexi cache mgmt) [Backend - CPU, SSD, Redis, GDS, Infinistore]

KV cache offloading: most-effective in long-context, high concurrency and resource-constrained env.



garak - LLM vulnerability scanner

Gigascale AI Factories — 10GW → OpenAI

Astra ? - Raghu Ramesha

OpenAI triton : DSL & compiler to write high-perf custom GPU using Python

Layer 1 Raw model → compiled model
TensorRT LLM
[plan file aka optimized engine]
[enables continuous batching so that LLM ain't idle]
high-perf cpp code talking to H100s

Layer 2 Triton (Dynamo Triton) Node / Server
- loads the plan file → opens http/gRPC ports
Triton sits on top of TRTLLM
↳ Req comes in : Triton manages priority queue
uses TRTLLM to do optimized math
App". server (NGINX/unicorn)

Layer 3 cluster / Fabric : Dynamo
Manages fleet of instances running Triton
[Basically a router]

→ Instance A has the memory (KV cache), so route there
NIXL — moves data b/w nodes
Load balancer + tg (VPC NW layer)

Instead of building 1 and 2 layer, use NIMs :)

Diff b/w continuous batching and dynamic batching :

Dynamic (Triton) Request-level packing
[Server level]
waits for specific window (5ms) / Spec # of reqs (16), Once met, sends all of them as a batch
10 words → short time
vs
500 words ↑↑↑ time
Tput / Latency ?

Continuous (TRTLLM)
↳ token-level packing
One req over
↳ insert next req.
GPU never idles
(2x / 4x >>> dyn batch)

Build a deep research agent

* LLM → RAG → Deep Research agent [summarize, reflect]

Agentic capabilities ē ~~agtic~~ RAG

Cons:- in general
- no access to on-prem data
- unavailable for work use
- hard to understand blackbox

~~RAG~~ Deep research

Agentic AI
- Nemotron model + Agentic reasoning + tool call

NB 1 { NVIDIA apikey on NAC
Tavily key for web search toolcall

NB 2 { NIM Deployment
Big model + more data → ↑↑↑ perf

---

pre-training | post-training | test-time compute

TT
PoT
PrT
Scaling → enhance acc. on domain spec data
QA / Inst. following

→ open book exam + add^n. time to CoT and do reasoning steps

~~Post~~ - Reasoning models for agentic AI app^n.

NB 3: Deploy RAG!
☆ Ingestion + retrieval / reranking + generation

NB 4: Deploy researcher.

Strawberry!

/no_think → no thinking

~~import Openai~~
from openai import OpenAI

client = OpenAI()
~~base_url~~ =
base_url =
api_key =

client.chat.completions.create( )
model =
messages = [{'role': , 'content

for chunk in ... :
if chunk.choices[0].delta.content is not None:
chunk.choices[0].delta.content is not None:
print(chunk.choices[0].delta.content, end="")

accesses 1st and only complete choice

- new txt fragment of this chunk
- prevents new-lines

Power of reasoning

Response quality → transparency → self correction → resp. length

understand docker flags:

docker run -it (-d detached) (--rm Remove cont when stopped) --name local-nim

container

--runtime=nvidia NVDA runtime for GPU access

--gpus all → allocate all GPUs

-p 8000 8000 → port for API access to cont

-v

-e → pass API key for model auth

--shm-size

↳ 20 GB shared mem for multi-GPU inf.

Cache model wt. to avoid redownloading

docker logs -f local-nim

name

integrations.api.nvidia.com/v1/chat/completions

api key not needed for local deploy.

0.0.0.0:8000/v1

docker stop local-nim.

NeMo retriever embedding NIM:

text → vector emb. for sem. search

Re-ranking NIM: Imp. retrieval acc. by re-ordering search results

NeMotron reasoning NIM: Reasoning model

Milvus vector DB: stores and searches document emb. @ scale

Doc. ingestion svc:

RAG query server:

1) Extraction phase – doc. ingestion + processing

vector emb. – capture semantic meaning of content in high dim space

(2 close / similar vectors)

2) Retrieval phase – query proc and resp. generation

search on semantic meaning than simple kw match.

Pipeline:-

Deploy NIM & NVIDIA endpt :

vector DB and orchestration services → LOCAL
(MILVUS) (RAG server / Ingestor)

LAYERS

Infra	Proc	RAG
- MILVUS standalone - MinIO (obj store) - etcd (metadata store) - Redis (cache for logs / chat hist.)	- NV-Ingest runtime (NIM doc-extr.) - Ingestor server (Doc. upload and Ingestion)	- RAG Server (Query/retrieval) - RAG playgrnd (web UI)

NIM local deployment –

```
ChatNVIDIA ( base_url=' ',).
    model=
    temperature =

NVIDIAEmbeddings (model = ' ',
docker run -d --name meta-llama3-8b-
                                instruct --gpus=
  -e NGC_API_key -v "$(LOCAL_NIM_
                    CACHE : /opt/nim/.cach
  -u $(id -u) -p 8000:8000
  nvcr.io/nim/meta/llam
                    -8b-
```

documents = []
for url in urls :
    doc = html_doc_loader(url) → // uses BS 4.
    documents.append(doc)

```
text_splitter = RecursiveTextCharacterSplitter
                (chunk_size =
                 chunk_overlap =
                 length_fn = len

                )
text = text_splitter.create_documents(documents)
metadata = [document[x].metadata]
index_docs (        )

model = NVIDIAEmbeddings(model = ' ',
                         truncate = 'END')

for doc in documents :
    text = splitter.split_text(doc.
                               page_content
    metadatas = [doc.metadata]

    if os.path.exists (  ) :
    update = FAISS.load_local (        )
    update.add_texts(texts,
                     metadatas = metadata
load_local
    update.save_local(  )

    FAISS.from_texts(texts, emb, metadata)
    docsearch.save_local (  )
```

~~Agentic AI:~~ Intro to RAPIDS :

~~Complex task~~ → ~~prom~~

Data loading + ETL	Model training + Analysis	Model Inf & Deployment
data acq. " cleaning " preproc	- Visualization - training - eval	- model deployment - " mangment
- [Rapids for cuDF]	- DL	- TRITON (Inf. server)
- Apache Spark	- cuML (RAPIDS)	- RAFT (vector search + ML)
- NeMo Data curator	cuGraph (graph analytics)	

data frame

Challenges →

```
df[['a','b']].value_counts().groupby('a')
```

indexes 'a' .head(1).

age sort_index(

.rest_index(

~~agg('max')~~ X

~~lambda x : x[~~

* - agg (lambda x : x.idxmax()[1])
  - agg (lambda x : x.idxmax()[1])

```
df[[a,b]].value_counts().reset_index()
    .agg('count': 'max',
         b: 'first')
    .sort_index()

df['Issue Date'].dt.weekday.map( )
    {'0': Mon ...}
df[' '].astype(datetime64[ns]')
['Issue_weekday'] =
    df.group(['issue_weekday']).['sum ~']
        .count(), sort_values()
```

%load_ext cudf.pandas

%%cudf.pandas.profile

" " ".line_profile

Series and DataFrame
replaced by proxy obj. that dispatch ops. to CuDF when possible

cudf doesn't support axis = 1 for count() method

```
df → df.groupby(' ').size().sort_index()
df.where(df[' '] == ' ').groupby(' ').size
```

⇒ (PC/C × 100) → pickup fraction

NVIDIA's genAI toolchain:

AI Frameworks & SDKs

- NVIDIA NeMo
- NVIDIA TAO Toolkit

opensource toolkit for dev-ing & FT of AI models

FT NVDA's pre-trained models c̄ your own data using transfer learning

RAPIDS

- Open source suite of GPU-acc. DS libs, speeding up dataprep & analytics on GPUs
- cuDF, cuML, Spark to speed up ETL

Inference and Deployment tools

- Tensor RT (low-latency, high tput inference on NVDA GPU)
- Triton Inf. server: Scale the model deployment
- Riva (GPU acc SDK for realtime AI App^n)
- NIM
- Kubernetes operator for deploying NIM microservices

Orchestration and mgmt

- GPU Operator (Kub) operator that automates NVDA driver mgmt / CUDA SW on ~~custom nodes~~ GPU workloads
- NGC Helm charts
- Base command (OS for DGX) cluster
- Fleet com. (Secure deploy, manage and scale AI app^n. across dist edge infra

- Deployment and Infra layer

★ DGX system (DL @ scale Supercomp)

★ SYS SW - NVDA driver and CUDA Runtime

- MLOps - AI workbench
  - MLFlow
  - Launchpad

DCGM / Prometheus / grafana for cluster monitor

NeMo guardrails :

SSN pattern : [0-9]{3}-[0-9]{2}-[0-9]{4}
ph.no. : [0-9]{10}

rails:
input: ← Flows, blocks, patterns } config patterns
flows:
- detect_pii

Operates on i/p & o/p

blocks:
detect_pii:
Intercepts unsafe req type regex on i/p side
patterns:
before LLM sees
- SSN pattern
- ph.no
action: block

Sanitize/block unsafe ~~req. before~~ gen from model

Clustering of Risks in customer service system

PII
-SSN
-ph.no
-email
reg.

Security bypass

Legal liability
- ~~of~~ draft lawsuit
- legal advice

Jailbreak attempts
Inst override, Role play

Harmful content

Fake reviews & competitor targeted o/p.

Financial fraud
- credit card data extraction
- Acc. takeover

Corporate Reg. espionage
- trade secrets
- pricing strategy

violation (HIPAA, GDPR)

- Structured metadata for each block

Devs and compliance officers can see why it was blocked

```
res = app.generate(...)
(res.explanations) → {'blocked': ..., 'reason': ...}
```

- Compliance audits
- debugging
- risk dashboards

Langchain equivalent

- PIIMiddleware
- ContentFilterMiddleware
- HumanInTheLoopMiddleware
- SafetyGuardrailMiddleware

after agent

Problems of AgentExecutor

- Blackbox
- state mgmt (can time travel to any step)
- Linear Loop

use this to convert string to a table → io.StringIO(generated_table)

Context free grammars

~~CC~~ CFG.from_string("""

```
json_schema.validate(instance=instance, schema=schema)
except exceptions.ValidationError as ve:
```

DAPT: Dask, RAPIDS and cuDN

Download + extract text → Lang det → Doc-level dedup → Qual filt

↓

Content filt / analysis

↓

Dataset const (Blend + Shuffle + save data for custom tokenization)

WIKI + ARXIV + GIT

unified rep. of all data
text + metadata

metrics for pre/post training

Dedup, domain correction, domain validation

—BOOST Acc.

* data diversity — MUST for diff. format, sources and dist. in domain

under-represented edge cases — present to prevent model bias

Synthetic generation of data (augmentation) creates diverse and balanced datasets

— Continuous data curation pipelines help maintain relevance

Arxiv, wiki, git →

Fuzzy semantic dedup
PII
word count / line count heuristic
unicode
save / shuffle to send to tokenizer

Downloader
Iterator
Extractor

Run — curation pipeline → text + pdf

Sequential class — add built-in + custom filter

↳ Read them jsonfile

DocumentDataset( )
.read_json

ds_text = pass original dataset text & code
ds_code =

tokenizer ready for model — training

curated

curation_steps_text = Sequential(
clean_and_unify,
ScoreFilter(TextLineCountFilter(),
text_field='file_type_count',
score_type='bool'),
filter_text)

Domain - adaptive PT.

- Learnt abt data curation

Sequential class

[Document Dataset]

jsonl → df. [Dask/Ray]

↓

unicode, PII, score, wordcount, Modify() → filtering/manip

Custom tokenization:

- compare vocabs of OG tokenizer & newly trained domain spec tokenizer
- download llama2 embedding model + tokenizer
- Expand OG tokenizer w new tokens to get extended tokenizer
- apply it to domain spec dataset, analyze user freq of new tokens and choose 95% cum freq
- init emb. of new tokens using OG tokenizer → tokenize new tokens using OG emb & o/p vecs are found by avg
- merge new vocab to OG table to get DAPT.

SFT:

* Enables PT model to specialize in a gn domain

Style
- train on labeled data
- preserve broad knowledge acq. in PT.

* updates large portion / all of model wt

* Improved task spec. perf
  - ↳ Needs relatively less data to train
  - ↳ Needs task spec. i/p → o/p dataset

* Few HW of modest HW.

Single SFT → multiple use-cases!

→ perf, cost & util.

Model customization:

FM → Foundation model

- Prompt learning
- SFT (teaches model to follow style)
- RLHF (user pref to ranking)

↓

use model as to how it's used

Fine tuning + eval

model → start from DAPT → of inst FT models
tokenizer → domain adapted tokenizer
Helpers → Data, tensor, pipeline

Loss fn → prod loss ~ optional regularization
(cross entropy) (if dio)

Learn Rate → Smaller than ~~DAPT~~ → unsupervised
Risk of catastrophic forgetting
Easy to overfit on small high Q curated SFT data

Overfit prevention → ~~Risk~~ early stopping ~ cross validation
MoE aux.

Eval → HF: (RLHF) → accurate fb.
- slow & collect feedback
- subjective bias

LLM as judge → quick
- biased by training data
- prompt sensitive
- hallucinate on domain spec qns.

B → 50% Tput H than
200
inc. mem & compute bw.

BEST FPR : SFT + LORA ✓✓✓

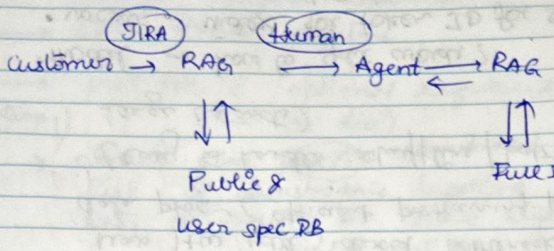
SFT:

JSONL → bin / idx → for faster access
good for storing structured data
and memory mgmt.
needed for high tput data loading!
- allows direct, randomized access to data samples
(↑↑↑ I/O ops., reduces mem. footprint
Max I/O util, avoid bottlenecks in data proc (efficient pretraining proc)
(easy to handle shuffling / batching of large datasets)

- model — how to split words?
- vocab — index for token ID for split words

Sizing LLM Inference Systems

Learning # of GPUs needed for LLM workflow



Q: How many GPUs are needed?

↳ gather requirements: * 3.5K words in 500 words out
* Nemo 43B GPT
* TTFT: 3s (limit)
* max: 31 req/s

Inference sizing: 13 DGX H100 systems
- Tput: 2.4 reqs/s
- latency for 1st token: 2606ms ≈ 2s (Prefill) [within limit specified]
- Inter-token latency:

Tput, latency, TCO metrics

Prefill: time b/w pressing enter and first o/p token (TTFT)
- Depends on # of i/p tokens
- populate KV cache for all tokens from query (or) prompt
- compute bound

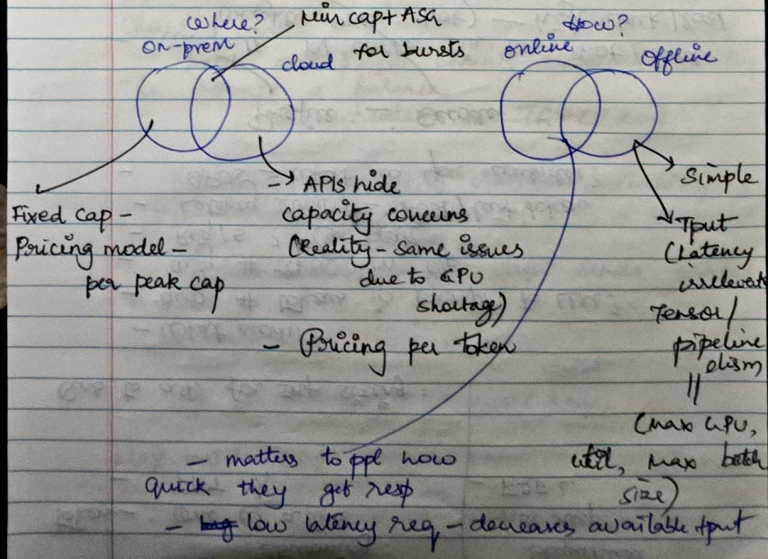
<20% of end to end

Decoding - memory bound
- generate resp word by word (and) token by token

80%

Streaming

INFERENCE SIZING:



	Streaming	Sequential
token	one @ a time	- full resp.
	- TTFT ✗	- E2E L

Qns. to ask for Inf. sizing :

- What model?
- avg # tokens in prompt to LLM?
- avg # token in O/p
- Req/s ? @ peak
- Latency limit - First/last token
- GPUs → what do you consider?

Prefill — Decode

- compute KV cache in one shot (load weights one shot) — high work/load

autoregressive process — one token @ a time

↳ load wts once — perform tiny mtx × vect → store in cache

high mem b/w use → compute to load wt for each token

- Decoding is serial by nature

(mem BW)

- Balance compute + memory bound

IFB : process prefill + decode in same batch

- nearly const. batch size for each token ↑↑ GPU util
- new req. exec quick as it waits for token gen ONLY, and not completion of cur. reqs

Chunked context : Separate prefill into chunks batch one prefill chunk + mult. decoding to attempt a balance

(implemented in TRTLLM)

- Irregular batch size introduce variability when measuring latency

If concurrency = 'c', client sends 'c' reqs in ||el

→ Maintains conc = 'c'

→ 'c' outgoing concurrent reqs.

Max. batch size :

Benefits of TP:

- N times low mem. footprint / GPU
- N times ↑↑ mem b/w
- xN — compute resources / model
- same precision, same acc.

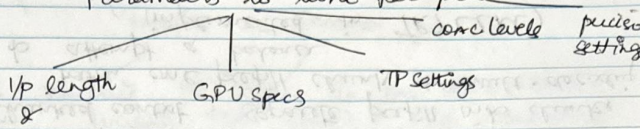
TP2: comm. overhead b/w GPUs

→ Need low-latency GPU interconnect

Reco. — NVLink enabled servers like DGX, HGX

— pairwise connected PCIe cards like H100 NVL

Parameters to tune for tput



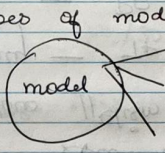
TTFT / E2E → low the ratio, ~~to~~ streaming is best

FP8 (since Hopper): — Reduced storage, inc tput & min. accuracy loss

- GenAI-perf + NIM benchmarking guide to measure performance
- models with >13 B params, use NVLink enabled systems
- Direct resp of LLM : stream
- Consumed by other LLM : seq.

cost and latency

Types of model parallism:



Data ||el:

- Each GPU gets full copy of model
- diff batch of data (for spec. gn)

* works iff model is small enuf. to be fit into a single GPU.

Goal: ↑↑ Tput.

Tensor ||elism: Reduces latency for 1 req. (split math → small comput)

Model is too big for a GPU. (communicate data to core)

Split math opn.

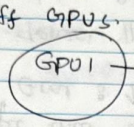
Split a tensor [1024 × 1024] by half

NVLink → GPU 1 calcs. top half, ... 2 calcs. bottom " → combine partial results before moving to next layer

Pipeline ||elism:

* If a model has 80 layers, we can't do TP
Talking b/w GPU is slow

* Split the model layers [1-20, 21-40...] to diff GPUs.



FIT MASSIVE models like GPT across GPUs ē cat saturating b/w of comm. like TP (fast networking)

Sit chumma

(PIPELINE BUBBLE)

Adv: model cap.

How ||elism works?

low latency — TP ē in single server (NVLink)

model cap — PP across multi servers to fit big model

— DP across cluster to accommodate 999↑ users (concurrently)

For a 175 B parameter model, start with

175B × 2 byte (FP16/BF16) = 350 GB
BIT

NVIDIA H100 ⟶ 80 GB HBM3 (High BW mem)

350/80 ≅ > 4 GPU ≈ 5 GPUs

~~DGX~~ DGX H100 node (8 GPUs)

TP: → 350/8 : 43 GB/GPU

FITS into H100 • 37 GB for KV cache

context window of 2048 ← 175 GB model

25 DGX nodes total

1-2 GB KV cache

≅ 25 users conc per DGX node

need 4pt DGX nodes

Quantize model wt → 350 → 175 GB

only 22 GB for wt: 58 GB for KV cache

Eval and customization of LLM :-

why eval LLMs :-
- Boost user trust
- max^m. acc & reliability
- enhance safety and compliance
- prevent AI hallucination
- adapt to local lang

(*) Cloud (vs) on-prem
(*) eval error rate and ensure fairness

How eval helps?
- data driven insights
- error analysis

Trade-off b/w cost, latency and quali

Validate ethical / bias eval over time

- Controlled eval - X
- Human-centered eval (HITL) - Human In The Loop
  ↳ relevance + language / tone
- Field eval
  ↳ A/B test, resp latency, error rate
  ~~Eval~~ ~~User~~ user satisfaction, helpfulness

Controlled eval — Text classification

Language modeling & generation — QA — Machine Translation — Conv. context understand

- Dialogue gen/creative writing

Metric:

Quan:

QA: Acc, prec, recall, F1 score

LM metric: perplexity - how much the pred is surprising (low → better)

BLEU/ROUGE → mach trans 'n' gram overlap (summary acc of ROUGE ~ human)

Academic/benchmark scores:

- GLUE
- Squad
- MMLU & GSM8K (grade school math 8K math prob solving 8.5K open ended math word prob.)

Qualitative/hybrid metric:

- Human judgements and LLM as a judge clarity, coherence, factual correctness

(Pairwise comp, rank o/p, open-ended tasks)

★ Task spec. ~~copy~~ qual: coherence, relevance

★ Model Robustness/Bias checks
↳ consistency, bias/fairness checks

Custom LLM eval

- uncover domain spec weakness/bias
- Make improved model selection
- Justify cost benefit and resource allocation

generalized claim → Specific claim

MMLU eval → massive multi-task language understanding
(evaluates gen knowledge & reasoning)

- MCQs (4/5 optn.)
- Factual acc, reasoning, dom spec und
- Diverse categories

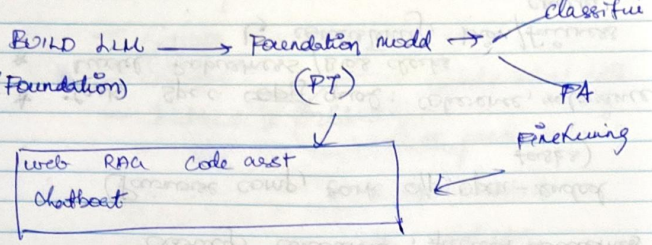
ans ē high prob in logits - CORRECT ANS

Prompt → model → logits → highest

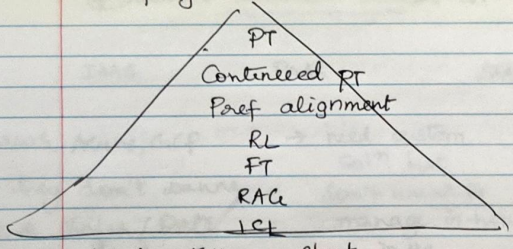
Why visualization matters?
Helps find trend

MLFlow! — Open-source platform
Log params, metrics, artifacts and models
Track eval across Runs

Light customization of LLM:
- Bridge domain gap
- Boost acc / reduce hallucination
- Naturalise tone / style
- enhance control / compliance



Adapting LLMs:



ICL: zero - shot

Few shot : prompt

SFT: - train model params on labeled task data
- highest task acc.
- exp compute / large chkpt

Domain adaptive FT:-
Continued PT

Deep-dive into PEFT
ⓐ backbone frozen - small # trainable param
ⓑ Techniques:
ⓐ adapter - MLP module b/w layers
ⓑ LORA - low rank decomp of wt. upd.
ⓒ P-tuning - cont. prompt prepended to key/val
ⓓ BitFit
↳ time only bias

Fast training, easy chkpt, simpler version ctl

Business rounds - senior SA.

IAAS | PAAS | SAAS

AWS, Azure, GCP

- They don't wanna have Infra / Data centre in-house

pros:
- agility in provisioning
- on-demand scalability
- management of AI/ML capabilities w/o in-house expertise

→ need custom soln. but don't want to manage in-house infra

Offer integration svc. and build custom soln.

Enterprise Resource Planning (ERP)

Cust R/n. Mgmt (CRM)

Consolidate that data into central data warehouse (Snowflake, Databricks)

→ servicenow, Data dog, Dynatrace

- communicate w many stakeholders in an org
- high financial commitment
- legal / regulatory policy complexity