Unsupervised phase - pre-training on large corpus of unlabeled text = **learning general-purpose representations** via **self-supervised learning** - next token prediction.

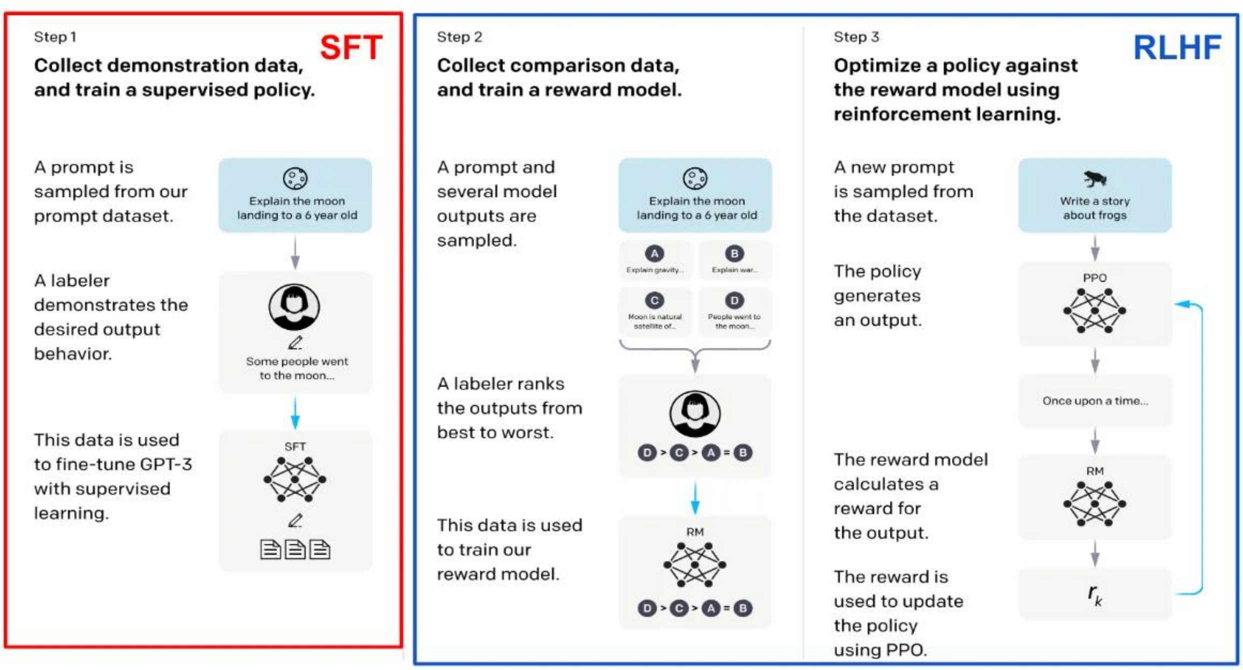
Supervised alignment phase

To **better align to end tasks** and user preferences. **Superficial Alignment Hypothesis** (SAH) = *most knowledge is learned during pre-training*, while alignment refines the delivery format.

- **Supervised Fine-Tuning** (SFT) teaches LLM to follow instructions by training it on **many examples of accurate responses to instruction-based prompts**. Good results require high-quality dataset.
- **Reinforcement Learning from Human Feedback** (RLHF) - optimizes the model using human-provided feedback: **multiple responses** *to the same prompts* are generated and **ranked by annotators** => **reward model** is trained with output = scalar reward is maximized (LLM is optimized) via the *Proximal Policy Optimization (PPO)* algo – SOTA for RL, special algos by OpenAI that utilize **policy gradient methods** - they *search the space of policies* rather than assigning values to state-action pairs. *Rejection sampling* is also used for RLHF (generates observations from a distribution).
- **Reward model** in RLHF takes *a prompt w/full chat history + response as input* and **predicts a preference score**, usually it's the **same architecture and weights as the LLM**, but its classification head (next token pred) is replaced with a **regression head** (preference estimation) and model is fine-tuned on preference data

Rejection sampling – LLM

- PPO takes 1 sample from model per iteration (iterative updates after each sample). Rejection sampling takes **K responses from LLM for each prompt**, scores each w/**reward model**, and **fine-tunes on best response** (Llama 2 70B used for rejection sampling to train all other models).
- **Rejection sampling fine-tuning** uses the same model (i.e., at the beginning of the RLHF round) to **generate an entire dataset of high-reward samples** that are used for fine-tuning in a similar manner to SFT (rejection sampling fine-tuning (**RFT**)).
- For best performance, RFT includes **best samples from all RLHF iterations**, not just current one.
- **Rejection sampling** for **first four rounds** of RLHF; *final round = RFT + PPO* sequentially (PPO last).



Proximal policy optimization (RL)

Trains agent to learn how to act in an env. in order to **maximize reward (**by OpenAI).

1. **Adjusts policy** (agent's strategy) by gradient ascent to maximize the reward.
2. Clipped **policy gradient objective** - prevents large updates - more stable and reliable improvement.
3. **Actor-Critic Method**: 'actor' updates policy based on feedback from 'critic' which evaluates the policy.
4. **Multiple Epochs per Update**: runs through data multiple times (epochs) - better sample efficiency.

Advantages: stable, reliable, efficient, simple. Drawback: HP sensitivity, computationally demanding.

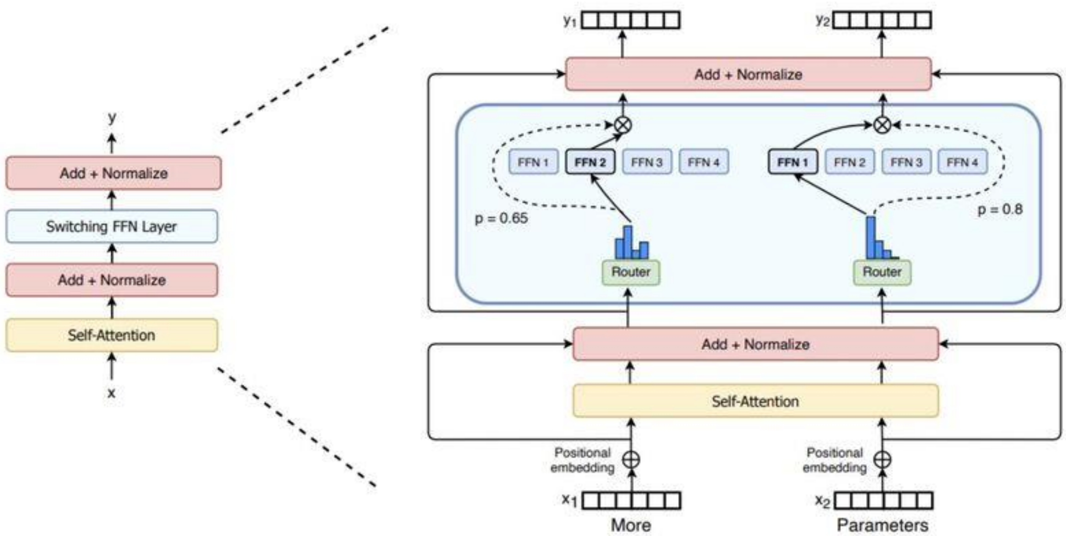
Mixture of Experts (MOE)

MoEs **replace the FFN layers** of transformer **w/sparse MoE layers** = each is a router + multiple experts (e.g., 8), each being an FFN w/own parameters. **Router/gate network** w/learned params and pretrained with rest of the network, **selects which experts** to send tokens to by taking each token as input and producing a **proba distrib. over experts (softmax gating)**. Although a MoE might have *many parameters, only some are used for inference* => *much faster inference* compared to a dense model w/same # params + faster pre-training.

Mixtral 8x7B

- High-quality *sparse mixture of experts model (SMoE)*, decoder-only model, has *8 models with 7B params* (FFN layers = individual experts).
- *Outperforms Llama 2 70B and GPT-3.5,* 6x faster inference, context window = 32K.

GPT-4 = MoE model w/16 expert models, each with around 111B params, total 1.76T



Mixture-of-Agents (MoA) - LLMs are better together

Multi-layer architecture - each layer = **agent** using multiple LLMs of **varying strengths**. Each new agent-layer **uses outputs from prev. agent-layers** to generate response. **Aggregate-and-synthesize prompt** to integrate responses from diff. agents. LLMs generate better responses when using outputs from other models. MoA superior performance with a 65.1% on AlpacaEval vs. 57.5% by GPT-4o.

Trained LLM Uses

- Basic chat or using embeddings in downstream tasks
- **Further fine-tuning** on domain-specific data.
- **In-context learning** - textual prompts incl. **few-shot**, CoT, or **RAG**.
- **Agents**

LLM Enterprise Use Cases

- **Summarization**: product reviews, reports, articles, insights from unstructured data
- **Conversational AI –** customer service bots, contact center solutions, enterprise Q&A,
- **Writing Assistant –** writer's block, writing assistant
- **Knowledge Mining –** domain-specific research, social media trends, cross-functional insights
- **Software Development –** faster coding, debugging, autocompletion, documentation
- **Image Generation –** marketing, logotypes, images / videos for ads (happy cleaner)

Imitation learning - fine-tune LLM w/outputs from more powerful LLM (~distillation).

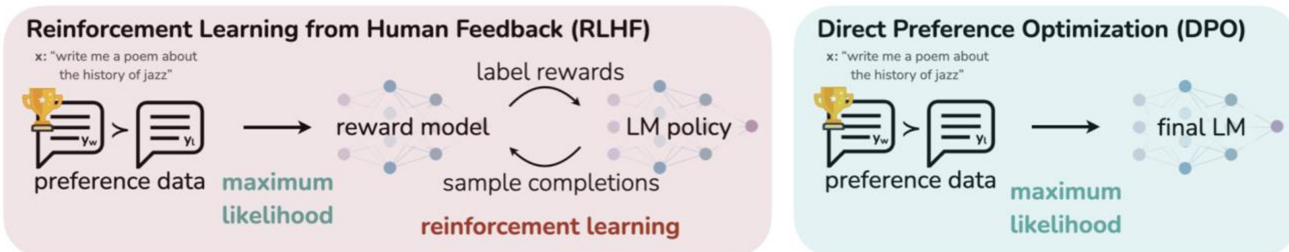
RLAIF - AI ranks the quality of several generated responses for each prompt during RL.

Online AI Feedback (OAIF) - no reward model.

Achieving alignment (SFT + RLHF) w/greater efficiency and **less human effort –** it collects real-time preference judgments from a **separate annotator LM evaluating pairs of responses** to determine which responses better align with goals. No reward model. *Easy customization* through flexible annotator prompting. Used when loss of precision from humans tolerated. Risk of bias or integrity issues => need *human oversight*.

Direct Preference Optimization (DPO) - stable, simpler, computationally **lightweight** algo, higher performing than RLHF. **RLHF - complex and unstable** *w/many hparams to tune* + **needs 3 LLMs**: fitting a **reward model** for human preferences (1st LLM), fine-tuning **LLM w/RL** to maximize the reward (2nd LLM), but making sure it is not too far from **original reference model** (RM = 3rd LLM); the latter is ensured by *KL divergence penalty to prevent reward hacking* (when LLM learns to output garbage that is rewarded highly like emojis or backslashes).

DPO users a single stage of policy training by **learning a policy directly from collected data** *without a reward model* significant hparam tuning – applying a *simple loss function optimized directly on a dataset of preferences* {(*x*, *yw*, *yl*)} = {(prompt, preferred, dispreferred responses)} = *classification problem* on the human preference data.



Identity Preference Optimization (IPO) - DPO tends to quickly overfit on preference dataset => IPO *adds a regularization term to DPO loss* to train models to convergence w/out early stopping.

Kahneman-Tversky Optimization (KTO) simplifies alignment & dispenses w/binary preferences - *defines a new HALO loss f(x) (Human-aware loss)* using **individual examples labelled as "good" or "bad"** (thumbs up / down) => uses an **adapted PPO for simplified offline, one-step alignment using existing preference dataset** - no update of reference model to enhance stability. Thumbs up /down **labels are much easier to acquire** in practice - e.g. **customer interaction data** to align LLMs to desirable outcomes (e.g., sales made). KTO matches or surpasses performance of DPO w/out relying on comparative preferences. HF **TRL** implemented DPO, IPO, and KTO in **DPOTrainer()**. **How much data -** 10-100K examples for SFT and 50K for DPO.

Odds Ratio Preference Optimization (ORPO)

Fine-tunes and aligns instruct LLMs (SFT + RLHF / DPO) **in a single step** - *no reward or SFT models* (ORPO - simpler than DPO and RLHF, performs on par w/DPO. Working with *preference data*, based on log odds ratio this method **introduces a penalty to the NLL loss f(x)** (negative log likelihood), **to favor generations in the chosen response sets**. ORPO shows faster training, lower memory, good results.

DPO vs. PPO. DPO - alternative to reward-based RLHF w/PPO because it doesn't require training a **separate reward model** - easier to implement; most of the LLMs on **top of public leaderboards** trained w/DPO, not PPO. But generally **PPO is better than DPO** (latter suffers from out-of-distribution data). U don't have to choose: Llama 3 training: pretraining -> SFT -> rejection sampling -> PPO -> DPO.

Simple Preference Optimization (SimPO)

New RLHF method **improves simplicity and training stability,** no reference model, similar to DPO, but **outperforms DPO** & ORPO on benchmarks + reduces time by ~20% & GPU memory by ~10% vs. DPO. Uses avg. log proba as reward & ensures **larger gap between chosen and rejected responses.** Built w/HF TRL.

Proxy tuning: a) fine-tune Llama 7B, b) **compute differences between weights** of fine-tuned and original Llama 7B, c) **add differences to Llama 70B** + normalized model outputs and generate the desired response.

PEFT = (Q)LORA

- Updates **small subset** of model's trainable params => much **faster, memory-efficient**.
- How it works: a) **identify layers** to apply LoRA to (k,v,q,output proj layers), b) **LoRA introduces and applies low-rank weights matrices to these layers –** these m. are much smaller in size vs. original weight matrices, significantly reducing the number of trainable params. *The original model weights are kept frozen, and only the low-rank matrices' parameters are updated during fine-tuning*.
- We can have **multiple lightweight portable LoRA models for various downstream tasks**
- **Matrix rank** – max # linearly **independent column vectors** or row vectors in the matrix (=info). Found using some distance measure like the **Frobenius norm**. Good for resource-constrained env. like Google Colab

rank = 5;   U, S, VT = **np.linalg.svd**(original_matrix, full_matrices=False)
# Keep only the top 'rank' singular values
U_k = U[:, :rank]; S_k = np.diag(S[:rank]); VT_k = VT[:rank, :]
# Construct low-rank matrix
low_rank_matrix = **np.dot(np.dot**(U_k, S_k), VT_k)

Quantization

Reduces weights' precision from a higher precision format (float32) to lower precision format (int8 or float16) and involves **mapping continuous range** of weight values **to a discrete set** of values *decreasing model's memory footprint and inference time + speeding up computations*; **trade-off in accuracy** - beneficial for deploying models on **resource-constrained** *devices* such as mobile phones or embedded systems. Example - your float32 weight = **0.34567** (range from -1 to 1); q. maps this continuous range to an 8-bit integer scale, e.g. from 0 to 255. => apply **scaling and rounding**, e.g. map -1 to 0 and 1 to 255 => 0.34567 becomes **172**.
LORAX (Lora Exchange) - serve large LLM once with many different adapters.

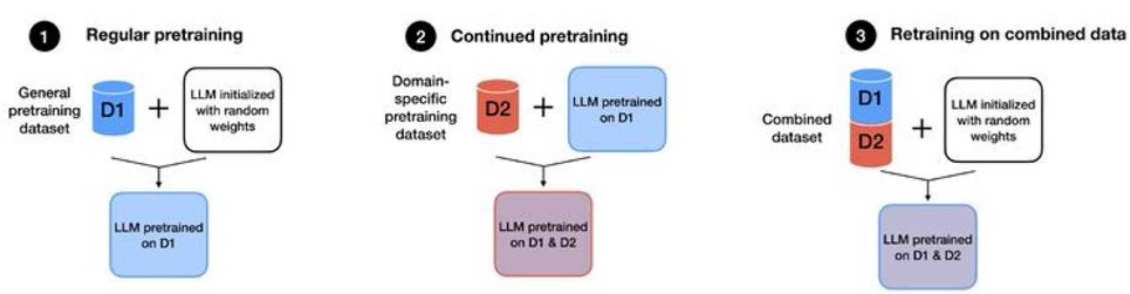
DORA - **weight-decomposed low-rank adaptation**; decomposes weight updates into a) *magnitude*, b) *direction*. *Direction* handled by *normal LoRA*, whereas the *magnitude* is handled by *separate learnable parameter*.

Galore - a) *gradients projected into a lower-dimensional (low-rank) subspace*, b) projected gradients, model weights **quantized** from 32-bit float to 8-bit int, c) these gradients are used to update model weights, d) weights are then de-quantized and updated.

Pruning + PEFT QLORA - 1) **up to 40%-50% of LLM layers pruned** with min. impact on accu, 2) **Identifying layers** to prune: via **similarity score** to find redundant or less important layers (lowest angular distance), 3) **pruning strategy**: progressively delete layers that showed min. change in output when compared to adjacent layers, 4) **fine-tune post-pruning** (small amount of **PEFT QLoRA)** to recover lost performance. **Deep layers** removed with negligible effect (not shallow ones) - inefficient training of deep layers?

Continual Pre-Training - common scenarios *new data arrives*:
1) *Regular pretraining*: initialize model w/random weights and pretrain on dataset D1
2) *Continued pretraining*: adopt pretrained model from 1) and pretrain it on dataset D2
3) *Retrain*: same as 1), but train on datasets **D1 + D2 –** 2x more expensive than continual pretraining.

Third option is commonly used, but continual pre-train saves significant compute. Concern w/2) *distribution shift* caused by new data may resulting in degraded performance on prev. data (*catastrophic forgetting)* or poor adaptation to new data. To combat: 1) **Re-warm and re-decay LR** (re-apply typical LR schedule), 2) **add small portion (5%) of original dataset D1** to new dataset (D2) to prevent catastrophic forgetting.



RAG Best Practices

RAG **grounds LLM in external knowledge** - *prevents hallucinated or incorrect information*. Deploying requires **extensive experimentation** *to tune / optimize many params*. Pay attention to:

- **Evaluation -** *component-wise* (retrieval, LLM's quality evaluated separately) or *end-to-end eval.* Key metrics: a) *Retrieval_Score*, b) *Quality_Score*. Need reference dataset, human scores.
- **Data quality** (inaccurate, biased data) data prep, diverse d., knowledge graph and addit. context / metadata
- **Chunking** - heavily *affects RAG* quality (more'n embeddings). Effective **chunk sizes of 100-700**, larger chunks - noise. **# chunks** - diminishing returns *beyond 7* due to LLM's context length. Strategies include using *smaller chunks* **and retrieving adjacent chunk** content or **chunk overlapping**.
- **Embeddings - off-the-shelf models** *work well often*, but **fine-tune** embed model on the *domain* if needed. *Smaller models* may outperform large ones.
- **Retrieval** – a) *term-based* (e.g. BM25), b) *vector similarity* (embeddings), c) *hybrid* retrieval.
- **LLM –** experiment for your *app's unique demands* - accuracy, latency, cost. Larger models better for *reasoning*, use efficient *Mixture of Experts* (Mistral outperforms Llama 2 70B), *intent classifiers* help map user query to predefined canonical forms.
- **Tune and** explore configurations, optimize hyperparameters

RAG Benefits

- **Up-to-date Info**: LLMs - fixed knowledge **cutoff dates**; RAG allows for easy **incorporation of current info** into LLM's output, *bypassing the limitations of finetuning* in updating LLM knowledge.
- **Data Privacy & Security**: safer *alternative to adding proprietary data in LLM train sets* (extraction attacks).
- **No overfitting and catastrophic forgetting** associated with fine-tuning
- **Reducing Hallucinations**: lowers the risk of incorrect responses by LLMs by providing direct access to reference data.
- **User Verification**: RAG **enables users to verify the output** *of LLM* by providing direct references to the data used in generating model outputs.
- **Ease of Implementation**: Compared to finetuning, RAG is **simpler and more cost-effective** to implement, with the possibility of enhancing retrieval model quality without the need to train the LLM itself. BUT COST OF PROMPTING. No MLE expertise to fine-tune.

RAG vs. Fine-Tuning

Fine-tuning increases accu by **6%**. *RAG* additionally increases accu by **5%** more. Choice of approach depends on specific **app**, nature and size of **data**, **available resources** for model development. Use either or both:

a. **Fine-tuning = model adaptation** - changing the *LLM's behavior (structure=weights), vocabulary, writing style, customizing model's tone or jargon* for a niche application, but not changing its knowledge (Google example);
b. **RAG relies on updated external data** to generate outputs **grounded to custom knowledge** while the LLM's vocab and writing style are unchanged,
c. *if your app needs both* custom knowledge & LLM adaptation - use a **hybrid approach** (RAG + fine-tuning),
d. *if you don't need either*, **prompt engineering** is the way to go.

Types of RAG: 1) **GraphRAG** (RAG w/knowledge graphs (KGs)) for augmenting context during generation w/structured domain-specific knowledge; includes automated KG construction (triple extraction), 2) **RAFT** trains special **Q&A model** w/CoT responses that is robust in ignoring irrelevant **distractor docs**, 3) **SELF-RAG** introducing self-reflection by fine-tuning LLM to predict if retrieval is needed & then evaluate relevance of retrieved info, 4) **Corrective RAG**: uses lightweight T5 model to *asses quality of initially retrieved docs* (classify as Correct, Ambig. - supplement w/web search, Incorrect - replace w/web) - more flexible, easy to implement than Self-RAG. 5) **HippoRAG** - converts text into schemaless KG in offline indexing phase; then retrieval.

Embedding Quantization: substantial cost and latency reductions in retrieval and similarity search + fewer bits for storage. Example: [12, 1, -100, 0.3,] => [1,1,0,1,] (0 if negative):

- **Binary Quantization**: 45x lower latency, 32x less memory, 96% of retrieval performance.
- **Scalar (int8) Quantization**: 4x lower latency, 4x less memory, 99.6% of retrieval performance.
- **Combine both**: binary search for min. latency & memory + scalar rescore for high performance = save costs.

Embedding Truncation: w/min. performance loss, **faster retrieval, clustering**, etc. Train model on domain.

AGENTS

AI agents transitioned from rule-based automation and can perform tasks autonomously. Learning, data-driven decision-making, continuous improvement. **Benefits:** efficiency and cost reduction, enhanced decision-making, scalability and adaptability (increasing workloads, respond quickly to market changes).

Use Cases Across Domains

- **Finance:** Automated trading, risk management, fraud detection.
- **Healthcare:** Diagnostics, patient management, personalized medicine.
- **Manufacturing:** Production optimization, supply chain management, predictive maintenance.
- **Education:** Personalized learning, administrative automation, intelligent tutoring.
- **Publishing:** Content creation, editorial processes, recommendation systems.

Agentic Workflows

Reflection: **automates the delivering of critical feedback** => *model automatically criticizes its own output* and improves its response (**prompt LLM for constructive criticism).** We can give LLM **tools to evaluate its output (**run code through *unit tests*) or *search the web* to double-check text output.

Tool Use: LLMs **perform complex, multi-step tasks** efficiently by **leveraging a variety of external functions and tools** to gather information, take actions, or manipulate data, extending capabilities. Examples: search different sources (web, Wikipedia, arXiv, etc.), interface w/productivity tools (email, read/write calendar, etc.) OR execute code. We can prompt LLM **describing what the desired function does** + input arguments => **LLM automatically chooses the right function**. Can hundreds of tools at LLM's diksposal.

Planning: LLM autonomously **determines a sequence of steps and tools to accomplish a complex task** that can't be completed in a single action. Example: transforming image of boy to girl in same pose OR **online research** task - break it down into subtopics, synthesize findings, compile a report

Multi-agent collaboration: **decomposing** complex tasks **into subtasks**, each **handled by different agents**, e.g. writing software => helps LLMs focus on smaller tasks. Different agents **prompt one or more LLMs for specific tasks**. Even if same LLM - different **focused prompts** to optimize performance.

Types of AI agents: 1. Creative Engines: new content, creativity, **2. Information Retrievers:** extract info from DBs, search engines, APIs, **3. Syntactic Operations:** grammar correction, rephrasing, summarization, translation. **4. Logic Engines:** break down complex tasks into logical steps and create action plans.

More Agents Is All You Need: more agents increase LLMs accuracy (sampling-and-voting technique). E.g. with 15 agents, Llama2-13B equals Llama2-70B.

Prompt Engineering

- **Automatic prompting**: LLM iterates on prompt & improves quality based on e.g. clf performance.
- **Chain of thought**: Each example **explains how the problem is solved step-by-step**: the *problem is broken into small parts that are solved individually*. Improves reasoning performance.
- **Chain of code** (outperforms CoT): *encourage LLM to format linguistic or arithmetic sub-tasks in a program* as flexible pseudocode - broadens the scope of problems LMs can tackle. Steps:
  - **Define** a linguistic or arithmetic reasoning task
  - **Code**: LM writes code / pseudocode to outline a solution.
  - **Emulation of code**: for non-executable parts of code, the LM emulates the expected outcome, simulating the code execution.
  - **Combining outputs**: LM combines code execution + emulation = comprehensive solution

WISER Framework

- **W** Who - Assign an identity or role
- **I** Instructions - Tell the model what to do
- **S** Sub-tasks - Break it down into simpler steps
- **E** Example - Provide examples (if applicable)
- **R** Review - Look at output / evaluation metrics and iterate

1. Write Clear and Specific Instructions: longer prompts w/context & details yield more accurate results.
- Use **Delimiters** for Clarity (triple backticks)
- Ask for **structured output** (JSON)
- Ask the model to **check conditions** (if then)
- **Provide examples** ("Few-shot" prompts)

2. **Give the model time to "think":** specify steps, let model think step by step before giving the final answer.
3. **Balance specificity with creativity** - be specific + let the model be creative: a) exhaustive details and context, b) **top p** - higher p increase randomness / creativity in next word prediction, lower p - next-token selection more predictable (default 1.0), c) **temperature –** higher temp. increases randomness / creativity (default 1.0)
4. "**Act as...**": extremely powerful.
5. **Always double-check** if hallucinations (e.g. ask for documentation **as proof**)
6. **Iterate** to find more efficient prompts (change words)
7. **Itemize instructions** (better to understand vs. long paragraphs)
8. **Avoid negations** (confuses model)
9. **Chain-of-thought** prompting

Retrieval Augmented Thoughts (RAT): RAG + COT ⇒ RAT - iterative CoT prompts w/info retrieval.
- Prompt LLM w/ zero-shot CoT (using RAG?)
- Retrieve information and each CoT reasoning step.
- Revise CoT steps based on the retrieved context.
- Generate a response using revised CoT steps and context.

RankPrompt: elicites high-quality feedback from LLM & enables LLMs to self-rank their responses using in-context learning, without additional resources: a) breaks down ranking problem into series of comparisons among responses, b) leverage the inherent capabilities of LLMs to generate chains of comparison as contextual exemplars, c) experiments w/11 arithmetic and commonsense reasoning tasks - enhances LLM reasoning.

Temperature Explained

Default=1.0. *Higher temperature* - *more creative, diverse outputs*, lower - more focused and deterministic generated text - *hyperparameter affecting proba distribution* of generated tokens generated by the GPT model. Mathematically, the temperature is **incorporated into the softmax function** (converts raw logits produced by the GPT model into a probability distribution):

$$p_i = \frac{\exp(z_i)}{\sum_{j=1}^{N} \exp(z_j)} \qquad p_i = \frac{\exp(x_i/\tau)}{\sum_{j=1}^{N} \exp(x_j/\tau)}$$

$$\sigma(\mathbf{z})_i = \frac{e^{\beta z_i}}{\sum_{j=1}^{K} e^{\beta z_j}} \text{ or } \sigma(\mathbf{z})_i = \frac{e^{-\beta z_i}}{\sum_{j=1}^{K} e^{-\beta z_j}} \text{ for } i = 1, \dots, K.$$

The parameter tau is called temperature and controls the softness of proba distribution. When tau gets lower, the biggest value in x gets more probability, *when tau gets larger - the proba will be split more evenly on different elements*.

Function Calling

Developers can now **describe functions** to LLM, and have the model intelligently *choose to output a JSON object* containing arguments to call those functions. Example: define *function for weather as json output = { 'city': city, 'state': state, 'country': country, 'temperature': degrees Farenheit}*. By providing schemas for "functions", the LLM will choose one and attempt to *output a response matching that schema*.