INF 385T · FALL 2026

TEXAS
The University of Texas at Austin

INF385T: Special Topics in Information Science: Language Models and Agentic Workflows

Week2: Structured Model Interfaces and Tool use

Dr. Abhijit Mishra

Week 1 Recap: From LLMs to Agents

DIRECT MODEL CALL
(Human / App provides the complete task)

DETERMINISTIC WORKFLOW
(Human written code determines the sequence of steps)

ADAPTIVE AGENTS
(System choses the next approved action from new observations)

Broader Takeaway: The model may be identical. What changes is the control architecture around it.

DIRECT MODEL CALL
(Human / App provides the complete task)

Example:

- Rewrite a course announcement in plain language.
- Human or application supplies the complete task context

DETERMINISTIC WORKFLOW
(Human written code determines the sequence of steps)

Example:

- Process an uploaded syllabus: extract the assignment deadline → validate the date format → save it to the course calendar → generate a confirmation.
- Code fixes the sequence of steps.

ADAPTIVE AGENTS
(System choses the next approved action from new observations)

Example:

- Investigate whether a student can graduate: choose which approved records and policies to consult next based on what each check reveals.
- Observations determine the next action.

Quiz: Direct call, Deterministic or Agentic

- **Q1:** A marketing manager asks a system to rewrite a product description in a playful tone.
  - A. Direct model call
  - B. Deterministic Workflow
  - C. Adaptive Agent

Answer: Direct model call (One bounded transformation)

Quiz: Direct call, Deterministic or Agentic

- **Q2:** An insurance system processes every claim using the same sequence: extract fields → check required documents → calculate eligibility → route for review.
  - A. Direct model call
  - B. Deterministic Workflow
  - C. Adaptive Agent

Answer: Deterministic Workflow (Predetermined processing stages)

Quiz: Direct call, Deterministic or Agentic

- **Q3:** A cybersecurity assistant investigates an alert, choosing the next approved diagnostic tool based on each result.
  - A. Direct model call
  - B. Deterministic Workflow
  - C. Adaptive Agent

Answer: Adaptive agent (Evidence changes the next action)

Quiz: Direct call, Deterministic or Agentic

- **Q4:** A publisher's system translates an article, checks it against a terminology list, and formats it for publication through a fixed pipeline.
  - A. Direct model call
  - B. Deterministic Workflow
  - C. Adaptive Agent

Answer: Deterministic Workflow (Predetermined processing stages)

Quiz: Direct call, Deterministic or Agentic

- **Q5:** A travel-disruption assistant monitors cancellations, explores available alternatives, and revises its plan as flight and hotel availability changes.
  - A. Direct model call
  - B. Deterministic Workflow
  - C. Adaptive Agent

Answer: Adaptive agent (The plan adapts to observations).

In summary

- No “free-lunch” with agants
- **Higher cost and latency:** Agents often require multiple model and tool calls.
- **Less predictability:** Adaptive paths are harder to test, reproduce, and debug.
- **Greater risk:** More autonomy increases the potential for errors and unintended actions.

In summary

- **Always Match complexity to need:** Use agents only when the next step must adapt.
- **Control cost and risk:** More autonomy means more latency, failure modes, and oversight.
- **Evaluate correctly:** Each architecture requires different tests, controls, and success criteria.

Agents should handle complex, adaptive tasks and not replace simple, predictable workflows.

An agentic system has several cooperating parts

No single component supplies capability, evidence, authority, and accountability by itself.

Week 2: Structured Model Inferences and Tool use

Roadmap:

- Transformers, LLMs, In context learning recap
- Structured outputs and schemas
- Function calling and tool use interfaces
- Model limitations and failure modes
- Tool descriptions as interface contracts

Announcement

- Frist Assignment posted
  - Objective: Build and evaluate a reliable, schema-validated model-to-tool interface that handles valid and adversarial requests, logs interactions, and improves through failure analysis.
- Project group formation
  - https://utexas.instructure.com/courses/1454968/assignments/7800078
  - Form here: https://forms.gle/gmB16NsJDUZnGhxi6
- Deadline: Next Thursday (09/10)

Transformers, LLMs, In context learning recap

What are Language Models?

- Models that estimate the joint probability distribution of words

$$W = (w_1, w_2, w_3, \ldots, w_n)$$

$$P(W|\theta) = P(w_1|\theta) \cdot P(w_2|w_1, \theta) \cdot P(w_3|w_1, w_2, \theta) \cdot \ldots \cdot P(w_n|w_1, w_2, \ldots, w_{n-1}, \theta)$$

Model parameters

Different ways to Model Languages

- **Masked LM objective (used in BERT, RoBERTa):**

$$\mathbf{argmax_w}\ \mathbf{p}(\mathbf{w_{MASK}} = ? \mid \{\mathbf{w_1}, \mathbf{w_2}, \ldots, \mathbf{w_N}\} - \{\mathbf{w_{MASK}}\}, \boldsymbol{\theta})$$

Training Example:

Input: "I want to <MASK> a movie tonight"

Output: "I want to watch a movie tonight"

- **Next Sentence Prediction (used in BERT)**

$$\mathbf{argmax}_{\mathbf{w}_1^{s2}, \mathbf{w}_2^{s2}, \ldots, \mathbf{w}_N^{s2}}\ \mathbf{p}\left(\mathbf{w}_1^{s2}, \mathbf{w}_2^{s2}, \ldots, \mathbf{w}_N^{s2} \mid \mathbf{w}_1^{s1}, \mathbf{w}_2^{s1}, \ldots, \mathbf{w}_N^{s1}, \boldsymbol{\theta}\right)$$

Training Example:

Input: "I want to watch a movie tonight"

Output: "May be Shawshank Redemption"

Different ways to Model Languages

- **Next word window prediction (used in GPT)**
  - Predict the next M words given the previous context

$$\mathbf{argmax}_{w_{i+1},w_{i+2},..,w_{i+M}}\ p(w_{i+1}, w_{i+2}, .., w_{i+M} |\ w_1, w_2, \dots, w_i, \theta)$$

Training Examples (M = 8):

Input: “A python function for calculating the square of a number is: ”

Output: “def calculate square (num) :”

Input: “A python function for calculating the square of a number is: def calculate square (num) :”

Output: \n \t square = num * num

Different ways to Model Languages

- **Denoising corrupted inputs (used in BART)**

$$\mathbf{argmax}_{\mathbf{w}_1^{correct},\mathbf{w}_2^{correct},\ldots,\mathbf{w}_N^{correct}} \mathbf{p}(\mathbf{w}_1^{correct},\mathbf{w}_2^{correct},\ldots,\mathbf{w}_N^{correct} \mid \mathbf{w}_1^{corrupt},\mathbf{w}_2^{corrupt},\ldots,\mathbf{w}_N^{corrupt},\boldsymbol{\theta})$$

Training Example:

Input: “quick dog jumped fox lazy a over a brown”

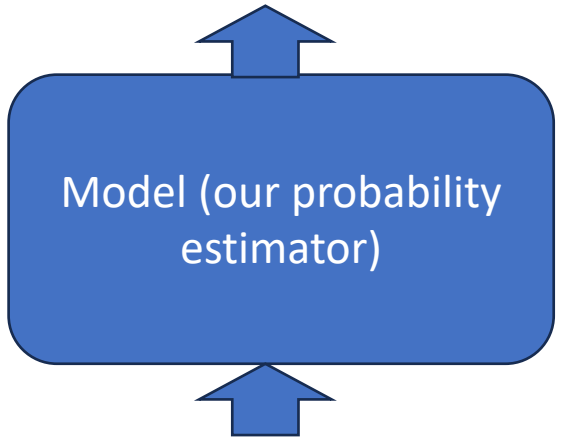
Output: “a quick brown fox jumped over a lazy dog”

Choice of Models = decide θ space

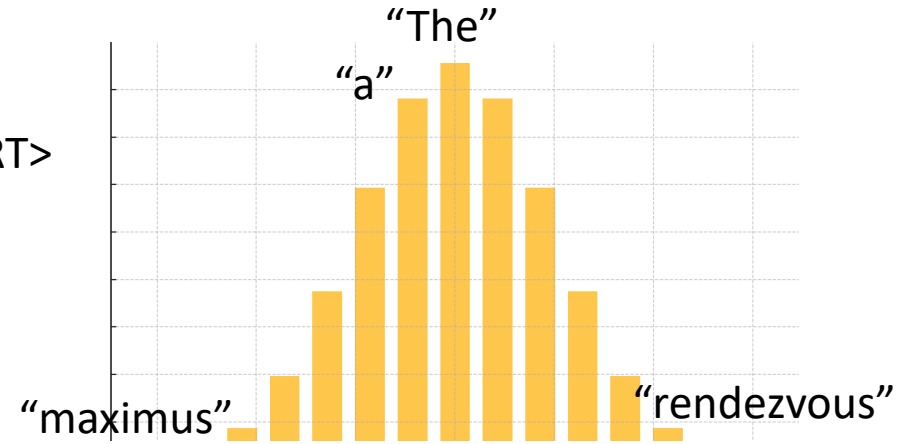
- Pre deep learning era:
  - θs estimated by simple “co-location” based probability estimation from large corpus
    - $\mathbf{p}\,(\text{”}\boldsymbol{teaches}\text{”} \mid \text{“}\mathbf{professor}\text{”},\ \text{“}\mathbf{The}\text{”}) \propto$ **How many times these words were present in the corpus**
    - Ex. SRILM, KenLM
- Deep learning (neural network based)
  - Recurrent Neural networks
  - Long Short Term Memory networks
- Modern:
  - Deep learning with high capacity networks such as transformers

Word prediction = a sequenctial classification problem

Estimates the probability of every word given <START>



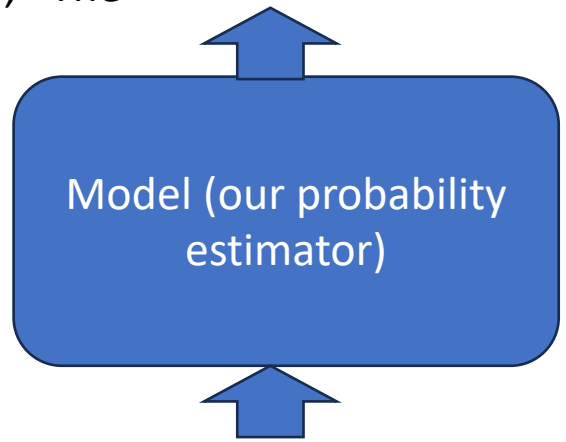
<START>



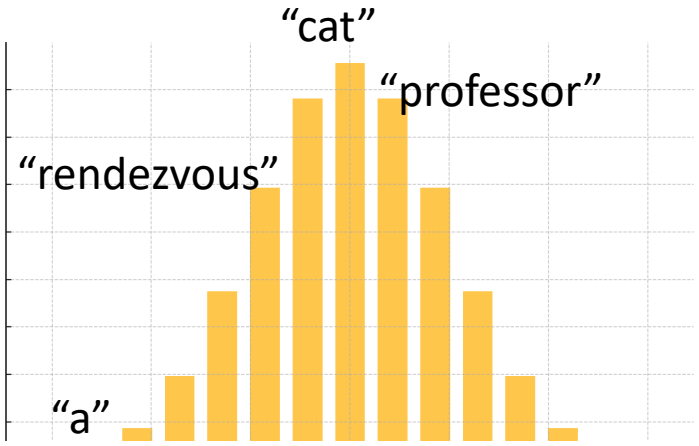
Picks “the” as the highest probable outcome

Word prediction = a sequenctial classification problem

Estimates the probability of every word given <START>, “The”



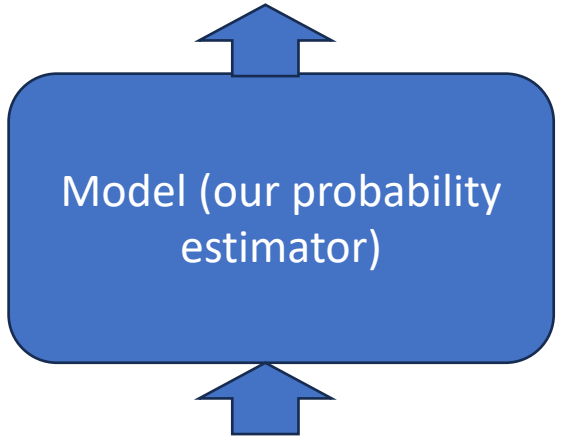
<START>, “The”



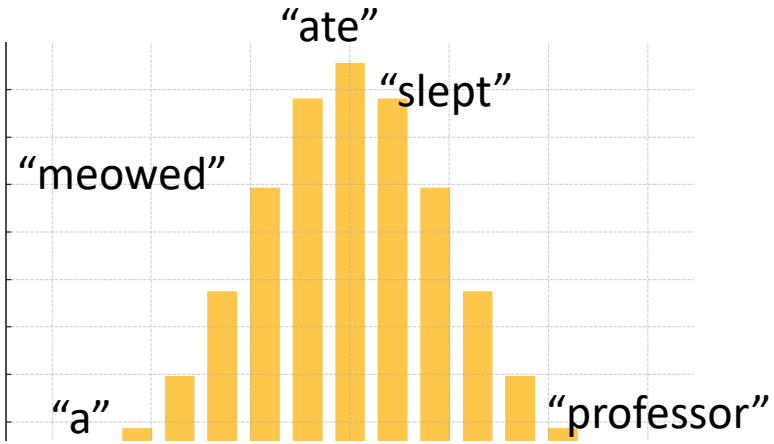
Notice how article “a” becomes least probable given the context

Word prediction = a sequenctial classification problem

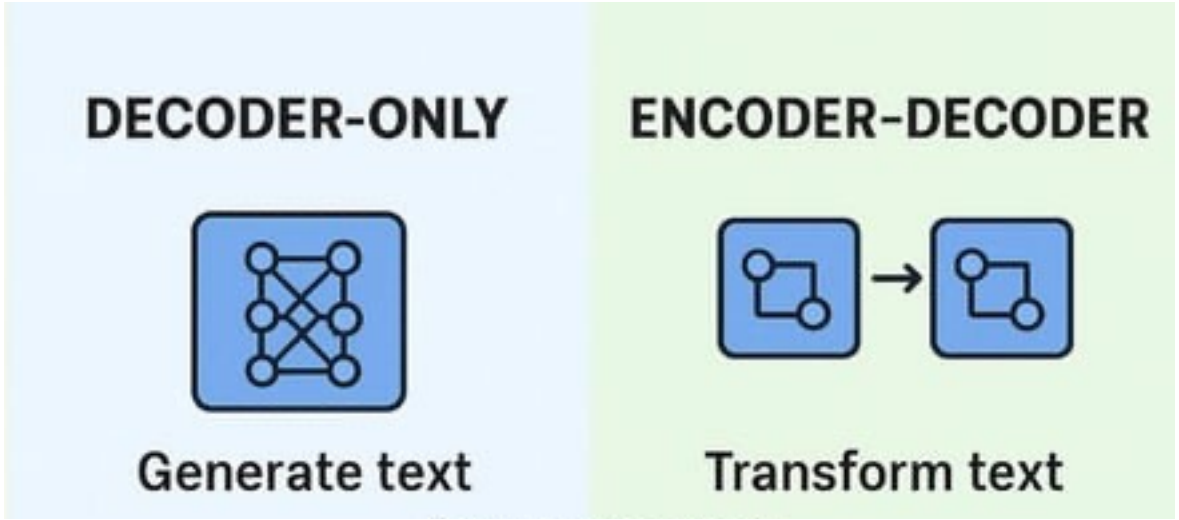
Estimates the probability of every word given <START>, “The”, cat”



<START>, “The”, “cat”



Different modern probability estimation models



Example: **GPT**
(Next word prediction)

Example: **BART/T5**
(Denoised input-output)

Let the input sequence be

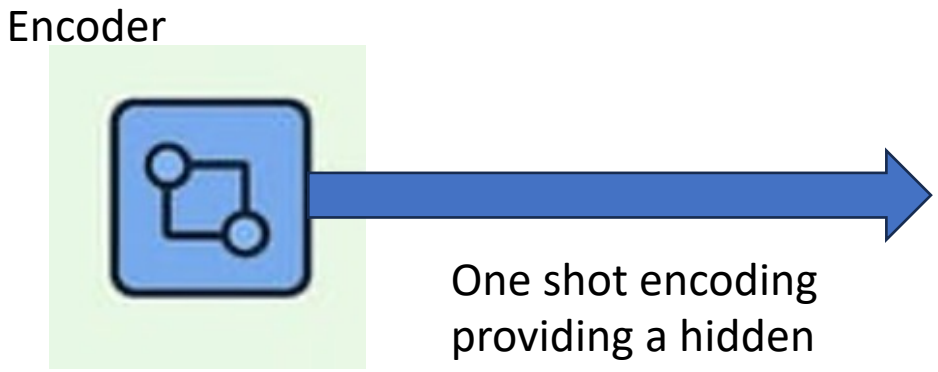
$$X = (x_1, \ldots, x_m)$$

and the output sequence be

$$Y = (y_1, \ldots, y_n).$$

Encoder decoder model

- Encodes the input sequence first, obtains a hidden representation

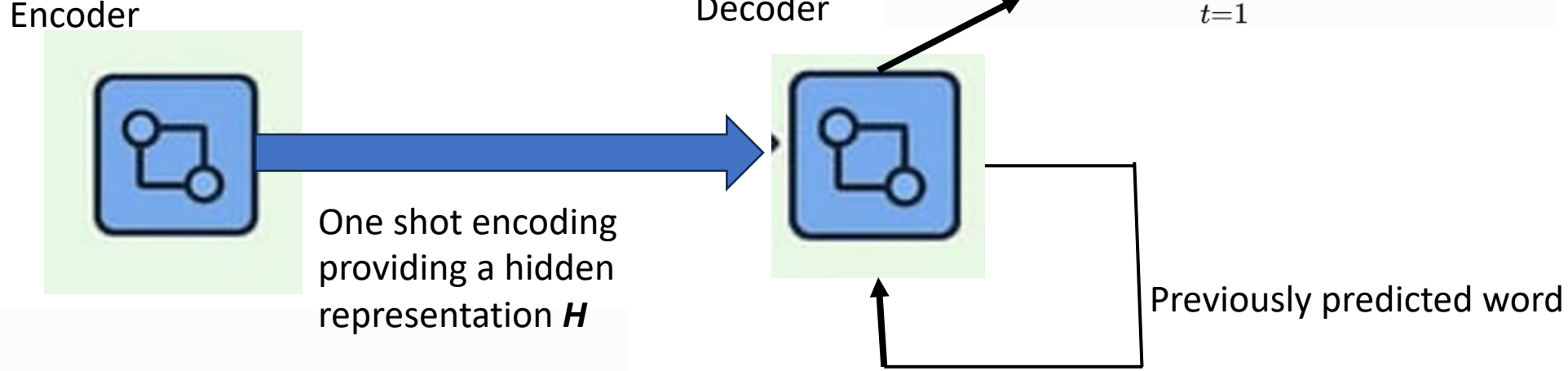


$$X = (x_1, \ldots, x_m)$$

Encoder-decoder model

- Encodes the input sequence first, obtains a hidden representation,
- Decoder uses it in the decoding loop long with previously generated words

$$P(Y \mid X, \theta) = \prod_{t=1}^{n} P(y_t \mid y_{<t}, H, \theta).$$



$$X = (x_1, \ldots, x_m)$$

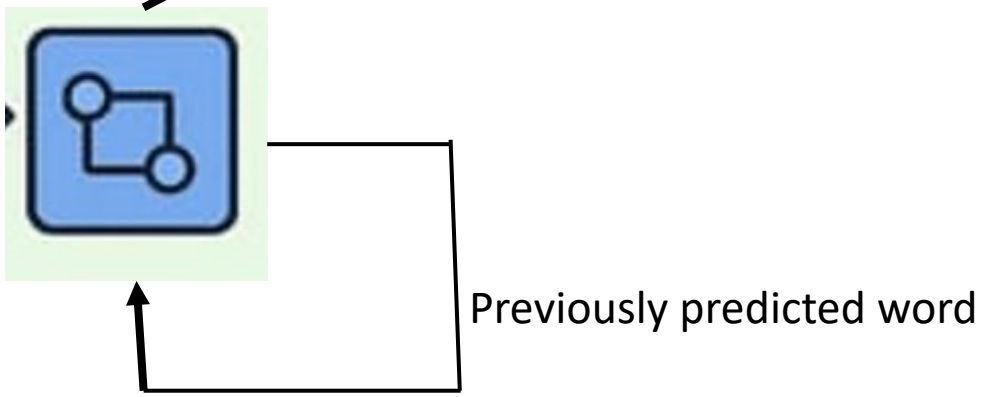
Practical usage and Downsides

- Text to text transformation tasks
  - Machine translation
  - Summarization
- Keeping encoding and decoding separate allows disentanglement of tasks
  - Understand English text and then translate to French
  - Works well with limited data setting
- Downsides:
  - Two separate components, more work to teach them well
  - Separate input and output processing
  - Less natural for continuation tasks
  - More complicated training and deployment

Decoder model (a.k.a Autoregressive models)

- Treat all tasks as completion tasks by concatenating inputs and outputs and train for predicting next words given previous

$$P(Y \mid X, \theta) = \prod_{t=1}^{n} P(y_t \mid X, y_{<t}, \theta).$$



Practical usage and Downsides

- All-purpose
  - Most task can be converted into “completion problems”
  - Example:
    - **Question answering:** The question and answer are treated as parts of a single text sequence. Given a question as the input prefix, the model frames question answering as an autoregressive completion task and generates the answer token by token.
- Downsides:
  - Sequential generation
  - One directional context
  - Same models need to learn to understand and generate (high capacity networks are need along with massive data.)
  - …

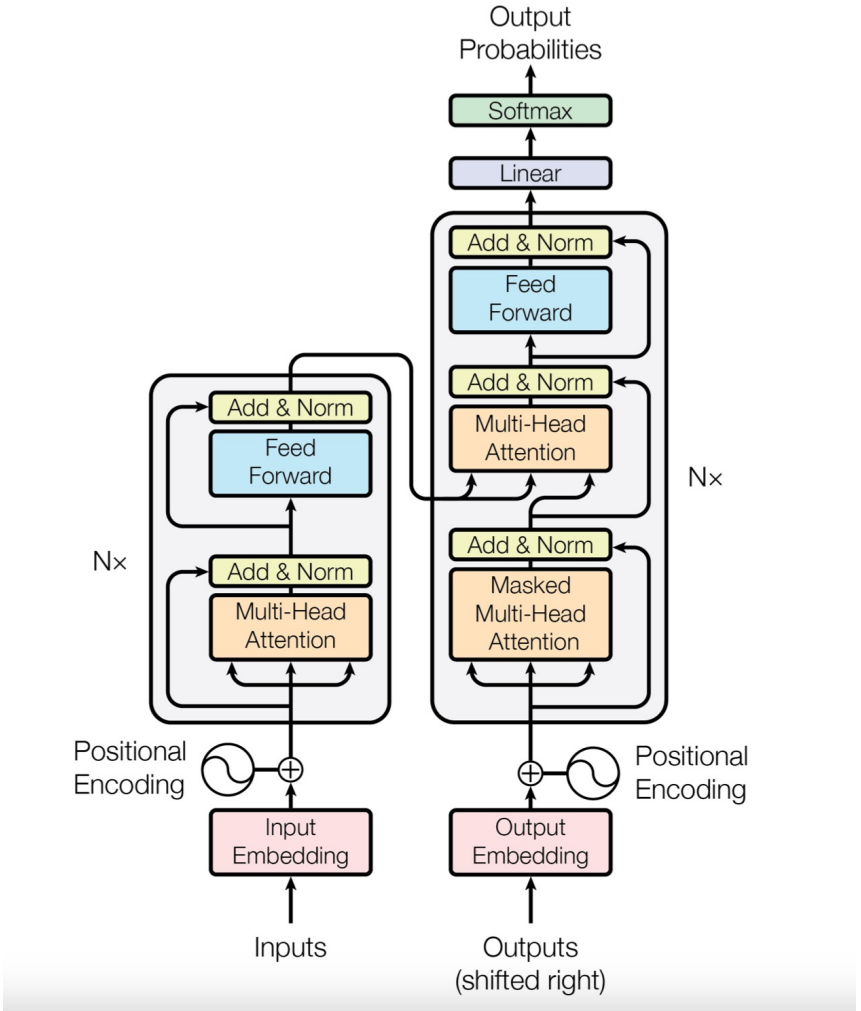
Transformer Case (Original model in "Attention is all you need paper")

- Model parameters / weights
- Encoder Decode model

All trainable layers in the transformers will have significant number of parameters

Significantly increased by adding more layers of encoders / decoders, increasing hidden layer size and max input length

OO Scope: Details covered in Deep Learning courses

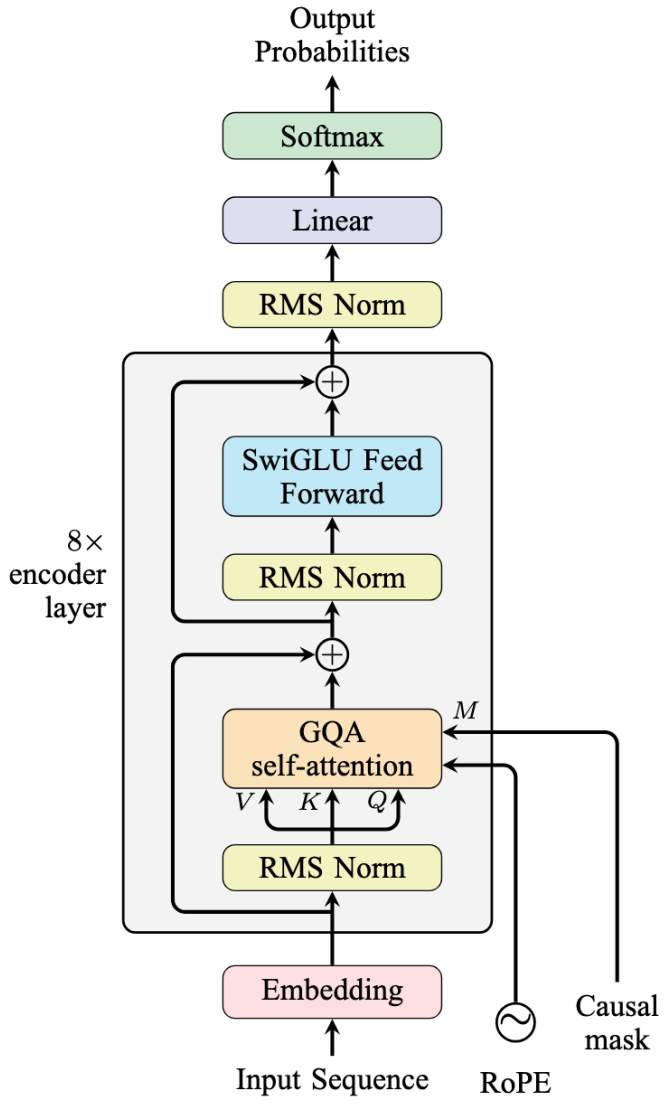


Decoder only variant

All trainable layers in the transformers will have significant number of parameters

Significantly increased by adding more layers of encoders / decoders, increasing hidden layer size and max input length

OO Scope: Details covered in Deep Learning courses



Data Requirements for building accurate probability estimators

- Large and diverse training datasets
- Many valid phrase and context combinations must be observed
- Sufficient examples of rare words and events
- Clean and representative data to estimate probabilities accurately

How to create large training data for text tasks?

- Process large amount of open web-text available
  - Wikipedia Dump, GeeksForGeeks, CommonCrawl, GitHub Data, StackOverflow
- Process the data accordingly to prepare input output
  - E.g., For Denoising LMs like BART, Implement noising of text to prepare input
- What file format:
  - Typically: Text files with one example per line
  - Large files are broken into chunks (e.g., database **shards)**
  - Big Data Formats: e..g, **parquet**

How to create large training data?

- Some basic text pre-processing steps
  - Cleaning data (removing non printable characters)
  - Redacting sensitive information (e.g., SSNs, Date of Births)
  - Tokenization *i.e.,* obtaining basic units of inputs from text, e.g., words, subwords,
- What platform is used:
  - Typically distributed computing infrastructure (Kubernetes based VMs, multiple nodes / computing units)
  - Speed up with **map-reduce paradigm**: File chunks are processed in parallel **(map),** results are combined later **(reduce)**

What happens during training?

- Loss / Error Functions :
  - Cross entropy loss across vocabulary V for the entire sequence M for all training examples in the batch B

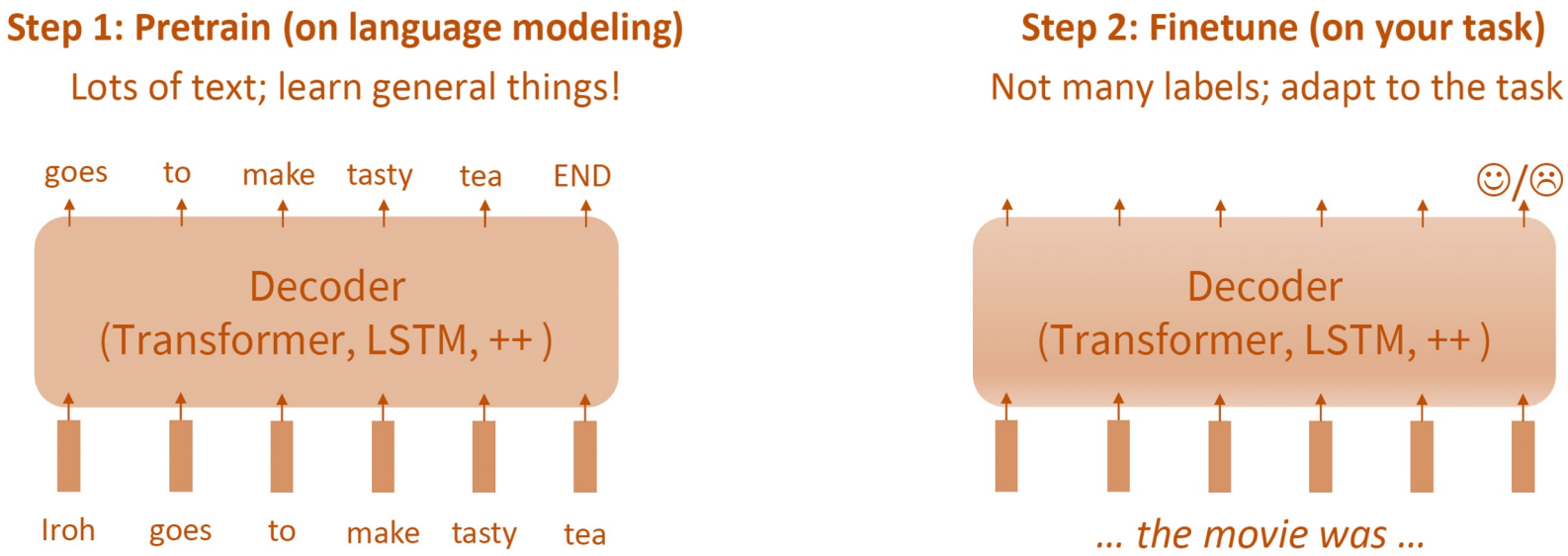
$$L(\theta) = \sum_{i=1}^{B}\sum_{j=1}^{M}\sum_{k=1}^{V} -w_{ijk}^{actual} \log(\mathrm{p}(w_{ijk} | w_{i1k}, w_{i2k}, \dots, w_{ij-1k}, \dots, w_{ij+1k}, \dots, w_{iMk}, \theta)$$

Neural networks undergo error back-propagation based training.

“Language modeling = a pre-training objective”

What are LMs actually useful for?

- **Fine tuning** on small amount of task specific data
- Pre-training with LM objective can improve NLP applications by serving as parameter initialization.



Why?

Consider Masked Language Modeling Objectve

Stanford University is located in _____, California. [Trivia]

I put ___ fork down on the table. [syntax]

The woman walked across the street, checking for traffic over ___ shoulder. [coreference]

I went to the ocean to see the fish, turtles, seals, and _____. [lexical semantics/topic]

Overall, the value I got from the two hours watching it was the sum total of the popcorn and the drink. The movie was ___. [sentiment]

Iroh went into the kitchen to make some tea. Standing next to Iroh, Zuko pondered his destiny. Zuko left the _____. [some reasoning – this is harder]

I was thinking about the sequence that goes 1, 1, 2, 3, 5, 8, 13, 21, ____ [some basic arithmetic; they don't learn the Fibonnaci sequence]

https://web.stanford.edu/class/cs224n/slides/cs224n-2023-lecture11-prompting-rlhf.pdf

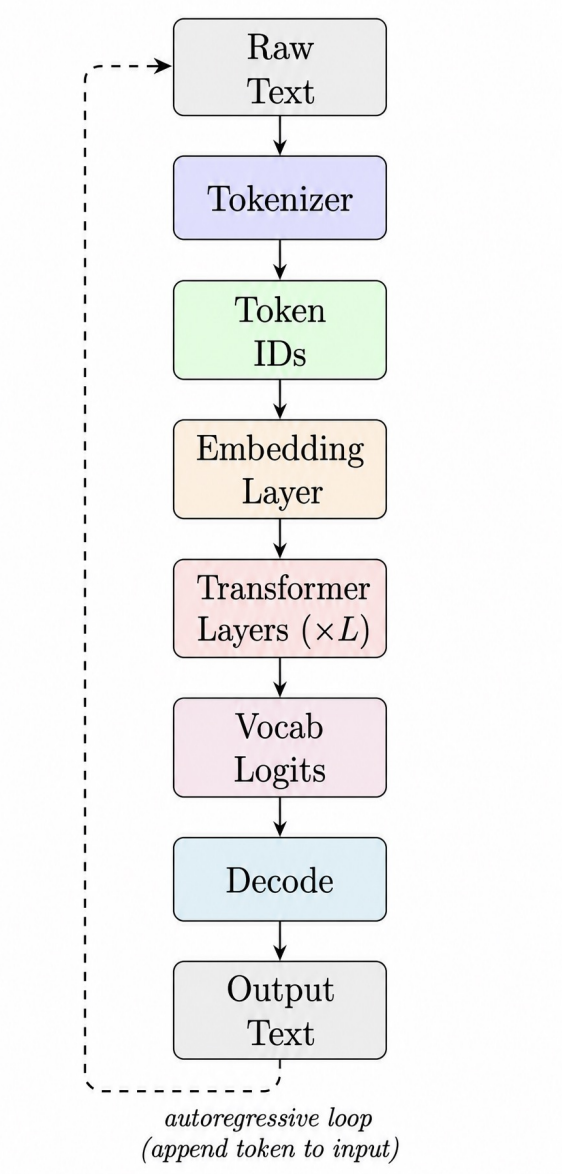
What are LLMs?

- Language models made larger by introducing many transformer layers
- Trained on massive web-scale data and fine tuned on human and agentic tasks
  - E.g., train on books and fine tune to produce JSON / Structured outcomes

LLMs

- **Scaling laws:** With increasing model size, training data, and computational resources, language-model performance improves predictably.

- **Emergent abilities:** New capabilities can appear unexpectedly when language models reach a sufficiently large scale
  - From GPT-2 to GPT-3, GPT-4, and GPT-5, increasing scale has revealed progressively stronger abilities
  - From coherent text generation to few-shot learning, complex reasoning, multimodal understanding, coding, and agentic tool use.



- **Tokenization:** Raw text is divided into subword units, not individual characters or necessarily complete words, using a learned vocabulary.
  - For example, “unhappiness” might be split into [“un”, “happiness”] or [“unhapp”, “iness”].
- **Embedding:** Each token ID is used to retrieve a learned dense vector from an embedding table. The vectors represent semantic information, so tokens with similar meanings often have similar representations.
- **Contextual Processing:** The transformer layers process the embeddings using self-attention, allowing each position to gather information from the permitted surrounding positions.
- **Prediction:** The final hidden representation is projected into logits over the complete vocabulary. These logits are converted into a probability distribution, and a decoding method selects the next token.

1 · From language modeling to reliable interfaces

A model predicts tokens—not truth

- Input text is converted into tokens.
- A transformer estimates the next-token distribution.
- Decoding turns probabilities into one output sequence.
- Fluent output can still be incomplete, inconsistent, or wrong.
- Software must therefore verify structure before use.

Example:

System: Please extract category and priority details

Input: “I was charged twice for my subscription.”

{“category”: “shipping”, “priority”: “pretty urgent”

- **Content error:** shipping contradicts the billing complaint.
- **Schema error:** urgent is not an allowed value (low, medium, or high).
- **Application response:** Reject the output instead of routing the ticket.
- **Syntax error**: Braces missing

Attention uses a finite context window

- Attention mechanisms in transformers link each token to relevant earlier tokens.
- The prompt supplies temporary task context and examples.
- Long or conflicting instructions can weaken reliability.
- The model may emphasize plausible wording over exact constraints.
- Clear contracts reduce the space of acceptable behavior.

Examples steer behavior

- Instructions and demonstrations steer the current response.
- Examples help most when they match the desired structure.
- One example shows the happy path; counterexamples expose boundaries.
- Validation is still required after the model responds.

System prompt:
Extract the category and priority from the user's support request.
User prompt:
"I was charged twice for my subscription."
Possible outcome:
This appears to be a billing issue that should receive prompt attention.

With a demonstration
System prompt:
Extract the category and priority. Return JSON only.
Example input: "My package arrived damaged."
Example output: {"category": "shipping", "priority": "medium"}
User prompt:
"I was charged twice for my subscription."
Possible outcome:

```
{
  "category": "billing",
  "priority": "medium"
}
```

2 · Structured outputs and schemas

A schema turns prose into an interface

- Required fields make missing information visible.
- Types prevent a date, count, or identifier becoming free text.
- Enumerations restrict categories to supported values.
- Nested objects preserve relationships between fields.
- Optional fields allow flexibility without weakening the core contract.

Running example: campus-event lookup

- **Student request: “Find an AI workshop on campus this Friday.”**
  The system must identify intent, collect date/category constraints, select a safe local tool, validate its arguments, and return a predictable response.
  We will reuse this same request throughout the lecture.

Campus-event response schema

```
EventResult:
    event_name: text                          # required
    status: found | not_found |
            needs_clarification               # fixed choices
    location: text                            # optional
    date: text                                # required
    category: workshop | talk | social        # fixed choices
```

Constraints stop predictable failures

- **required event_name** → prevents unnamed results.
- **enum status** → prevents unsupported states such as “maybe”.
- **optional location** → permits online or TBD events.
- **ISO date format (YY-MM-DDDD)** → avoids date ambiguity.
- **nested details** → keeps date and category attached to the event.

Validation is a gate, not a cleanup step

- **Parse:** Is the output valid JSON?
- **Validate:** Does it satisfy every field rule?
- **Decide:** Accept, ask for clarification, or surface the error.
- **Log:** Preserve the invalid output and validator message for analysis.

Do not silently repair every failure; visible errors are evidence for better design.

```
Define EventResult schema

Repeat up to 3 times:
    response = ask model for EventResult

    Parse response as JSON
    Validate response against EventResult

    If valid:
        If status = needs_clarification:
            ask user
        Else:
            accept result
        Stop

    If invalid:
        log output + validation error

After 3 failures:
    surface the error
```

```python
from typing import Literal
from openai import OpenAI
from pydantic import BaseModel, ValidationError

class EventResult(BaseModel):
    event_name: str
    status: Literal["found", "not_found", "needs_clarification"]
    location: str | None = None
    date: str
    category: Literal["workshop", "talk", "social"]

def validate(...):
  ...

client = OpenAI()

for attempt in range(3):
    response = client.responses.create(
        model="gpt-5-nano",
        instructions="Extract event information. Use YYYY-MM-DD dates.",
        input="The AI workshop is in Austin on September 18, 2026.",
        text={"format": {
            "type": "json_schema",
            "name": "event_result",
            "schema": EventResult.model_json_schema(),
            "strict": True,
        }},
    )

    try:
        event = validate(response.output_text, EventResult)
        break
    except ValidationError as error:
        log(response.output_text, error)
```

3 · Function calling and tool-use interfaces

What is a tool?

- A **tool** is an application-controlled capability that an LLM can request to retrieve information, perform computation, or take an action.

```python
def search_campus_events(query: str, date: str) -> list[dict]:
    """Return campus events matching a topic and ISO date."""
    ...
```

```python
def get_weather(city: str, date: str) -> dict:
    """Return the weather forecast for a city and date."""
    ...
```

```python
def lookup_order(order_id: str) -> dict:
    """Return the current status of an order."""
    ...
```

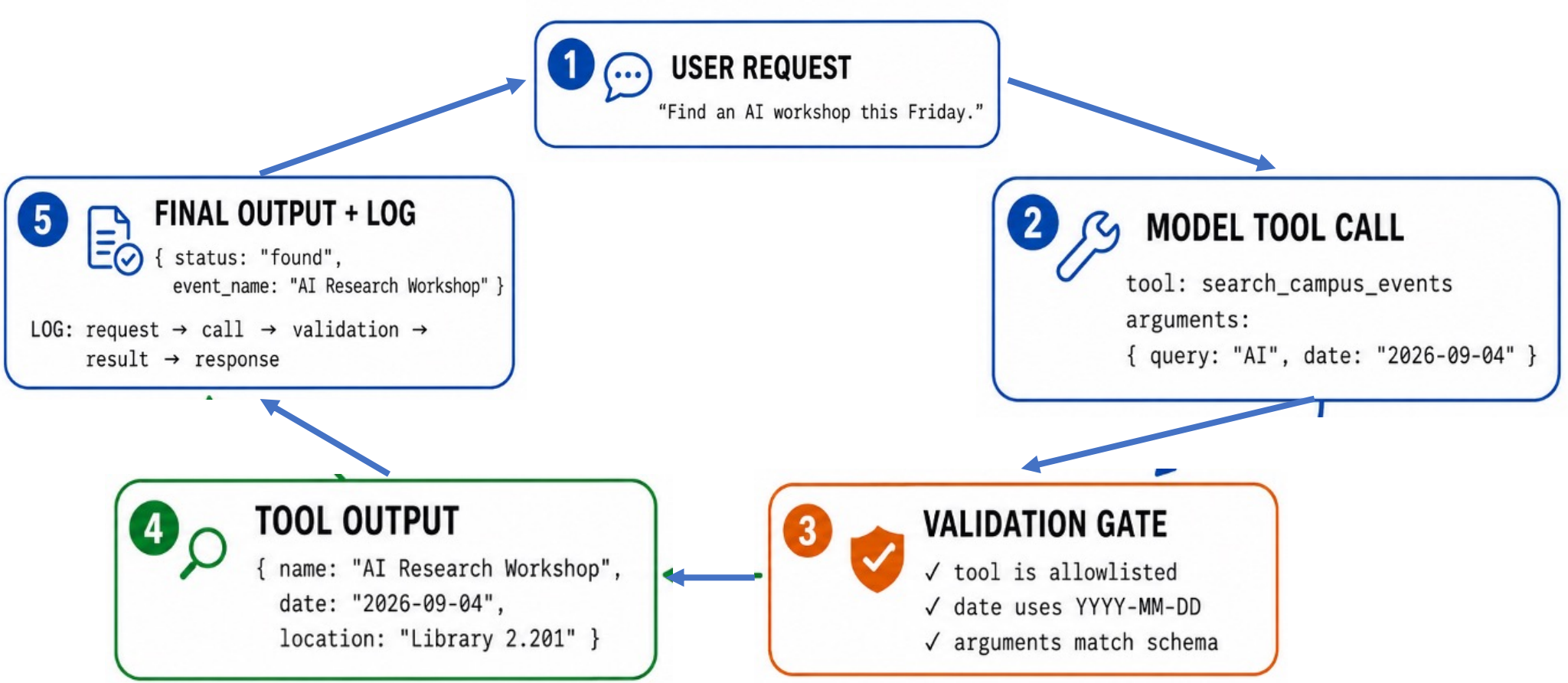
```python
def calculate_shipping(weight_kg: float, destination: str) -> dict:
    """Calculate available shipping options and prices."""
    ...


def search_documents(query: str, limit: int = 5) -> list[dict]:
    """Return documents relevant to a search query."""
    ...


def create_support_ticket(subject: str, description: str) -> dict:
    """Create a support ticket and return its identifier."""
    ...
```

A tool interface has five essential parts

- **Name**: unique and action-oriented.
- **Purpose**: when the model should select it.
- **Input schema**: typed fields and constraints.
- **Output structure**: stable data for the next step.
- **Error behavior**: explicit codes and recoverable guidance.

The model–tool–observation loop

User asks for an AI workshop this Friday.



4 · Model limitations and failure modes

Failures can occur at four boundaries

- **Intent**: the request is ambiguous or irrelevant.
- **Routing**: the wrong tool—or any tool—is selected.
- **Arguments**: required fields are missing or malformed.
- **Observation**: tool errors are ignored or misrepresented.
- **Response**: final JSON violates the response schema.

Adversarial case: ambiguity and missing information

- **Request: “Find something interesting on Friday.”**

Q: What kind of tool call should happen?

Adversarial case: ambiguity and missing information

- **Request: “Find something interesting on Friday.”**

  Problem: category and meaning of “interesting” are undefined.

  Correct behavior: do not invent preferences and do not call the tool yet.

  Return status = needs_clarification and ask one focused question.

Invalid case: malformed tool arguments

- **Proposed call: search_campus_events(date="Friday", category="seminar")**

  Validation failures:
  - date is not ISO format YYYY-MM-DD
  - “seminar” is outside the allowed category enum

  Correct behavior: expose both errors; do not execute the tool.

Reliable systems sometimes abstain

- **Irrelevant: “Write a poem about campus.” → answer directly.**
- **No lookup: “What does status mean?” → explain directly.**
- **Out of domain: “Book the event.” → do not take action.**
- **Tool use is a decision, not a default.**

Correct abstention is part of reliability.

5 · Tool descriptions as interface contracts

A strong contract answers six questions

- What does the tool do?
- When should it be called?
- When should it not be called?
- Which arguments are required and valid?
- What exact output can callers expect?
- How are empty results and errors represented?

Descriptions guide routing decisions

```python
def lookup_order(query: str) -> dict:
    """Gets information about an order."""
    ...
```

Weak

```python
def lookup_order(order_id: str) -> OrderLookupResult:
    """
    Retrieve the current status of one order.

    Use when:
    - The user asks about an existing order.
    - A valid order ID is available.

    Do not use when:
    - The user wants to cancel or modify an order.
    - The order ID is missing or ambiguous.

    Argument:
    - order_id: Required; format ORD-########.

    Returns:
    - order_id, status, estimated_delivery, tracking_url

    Empty result:
    - status="not_found"

    Error:
    - error_code and a safe error_message
    """
    ...
```

Strong

Error behavior is part of the contract

Example:

INVALID_DATE → identify expected format; do not execute.
INVALID_CATEGORY → return allowed values; do not substitute silently.
NO_MATCHES → return an empty list and count = 0.
NEEDS_CLARIFICATION → ask for the single missing decision.
Every error should support a deterministic next step.

Revision improves behavior at the boundary

- **Before: “Searches events.”**

  Observed failure: model calls the tool with date="Friday" and category="seminar".

  After: description requires ISO dates, lists the enum, and says not to call until required fields are known.

  Rerun: model requests clarification or supplies schema-valid arguments.

Week Summary: enforce the boundaries

- **Models generate candidates; schemas define acceptable structure.**
- **Tool contracts define capabilities and boundaries.**
- **Validation blocks malformed calls and responses.**
- **Logs and adversarial tests reveal what to revise.**

Design the interface so incorrect behavior is visible, bounded, and recoverable.

Over to tutorial