Large Language Models — Topper-Style Notes

- **Why it matters**: Automate text/code tasks, extract structure, augment apps.
- **Cost note**: Use `aipipe.org` proxy; monthly $1 allowance (course); do not exceed.

Using AI Pipe (OpenRouter + OpenAI via proxy)

- **Base URLs**:
  - OpenRouter: `https://aipipe.org/openrouter/v1`
  - OpenAI: `https://aipipe.org/openai/v1`
- **Auth**: `AIPIPE_TOKEN` replaces `OPENAI_API_KEY`.

Setup process:

1. Replace `OPENAI_BASE_URL` with `https://aipipe.org/openrouter/v1...` or `https://aipipe.org/openai/v1...`
2. Replace `OPENAI_API_KEY` with `AIPIPE_TOKEN`
3. Replace model names, e.g., `gpt-4.1-nano` with `openai/gpt-4.1-nano`

```
 curl https://aipipe.org/openrouter/v1/chat/completions \
  -H 'Content-Type: application/json' \
  -H "Authorization: Bearer $AIPIPE_TOKEN" \
  -d '{
    "model": "google/gemini-2.0-flash-lite-001",
    "messages": [{"role": "user", "content": "What is 2 + 2?"}]
  }'

curl https://aipipe.org/openai/v1/embeddings \
  -H 'Content-Type: application/json' \
  -H "Authorization: Bearer $AIPIPE_TOKEN" \
  -d '{"model": "text-embedding-3-small", "input": "What is 2 + 2?"}'
```

Using llm CLI:

```
 llm keys set openai --value $AIPIPE_TOKEN

export OPENAI_BASE_URL=https://aipipe.org/openrouter/v1
llm 'What is 2 + 2?' -m openrouter/google/gemini-2.0-flash-lite-001
```

```
export OPENAI_BASE_URL=https://aipipe.org/openai/v1
llm embed -c 'What is 2 + 2' -m 3-small
```

- Flex processing (50% discount, slower): add `"service_tier": "flex"`.
- `llm` CLI switching via `OPENAI_BASE_URL`.

Prompt Engineering (practical tactics)

- **Think of LLM as a smart colleague with amnesia** → give full context.
- **Best practices** (Anthropic, Google, OpenAI):
  - **Be clear and detailed**: specify audience, scope, constraints.
  - **Give examples (few-shot)**: 2–3 samples of desired pattern.
  - **Think step by step**: instruct reasoning before answer.
  - **Assign a role/persona**: “You are a senior data engineer…”.
  - **Use XML delimiters**: separate instructions, data, constraints.
  - **Ask for Markdown/JSON output**: readable or machine-parseable.
  - **Prefer Yes/No with reasoning** when applicable.
  - **Reason first, then answer** to reduce shallow justifications.
  - **Use proper spelling/grammar**.

Use prompt optimizers:

- Anthropic Prompt Optimizer
- OpenAI Prompt Generation
- Google AI-powered prompt writing tools

Examples:

```
 <role>You are a senior data engineer.</role>
<context>We store metrics daily in a SQLite DB.</context>
<task>Design a table schema and a weekly aggregation SQL query.</task>
<output>Provide Markdown sections and a final JSON object with fields: sche
```

```
 {
  "schema": "CREATE TABLE metrics (date TEXT, name TEXT, value REAL)",
  "query": "SELECT strftime('%Y-%W', date) w, name, SUM(value) v FROM metri
}
```

Key tactics:

- **Be clear, direct, and detailed**: Include all necessary context, goals, and details
- **Give examples**: Provide 2-3 relevant examples to guide the model
- **Think step by step**: Instruct the model to reason through problems step by step
- **Assign a role**: Specify a role or persona for context and style
- **Use XML to structure**: Use XML tags to separate different parts of the prompt
- **Use Markdown for formatting**: Encourage structured, readable output
- **Use JSON for machine-readable output**: When you need structured data
- **Prefer Yes/No answers**: Convert rating questions into binary choices
- **Ask for reason first**: Instruct reasoning before final answer

TDS TA Instructions and GPT Reviewer

- **TDS TA**: Virtual assistant trained on course content to help with doubts
- **Content creation**: Uses course repository and evaluation links
- **GPT Reviewer**: Technical content reviewer for correctness, clarity, and conciseness

TDS TA setup:

```
 # Clone the course repository
git clone https://github.com/sanand0/tools-in-data-science-public.git
cd tools-in-data-science-public

# Create a prompt file for the TA
PYTHONUTF8=1 uvx files-to-prompt --cxml *.md -o tds-content.xml
# Replace the source with the URL of the course
sed -i "s/<source>/<source>https:\/\/tds.s-anand.net\/#\//g" tds-content.xm
```

TA Instructions:

- Paraphrase unclear questions
- Cite relevant sections from course content
- Search online for additional answers with citations
- Think step-by-step and solve in simple language

- Ask follow-up questions to help learning

Content Review Process:

- Check for correctness and consistency
- Ensure clarity and approachability for high school level
- Assess conciseness and remove verbosity
- Provide actionable improvement suggestions

Structured outputs and logging

- **Schemas**: enforce JSON shape; validate in code.
- **Logging**: store prompts/responses (e.g., `llm` logs in SQLite; browse via Datasette).

LLM Sentiment Analysis and Text Extraction

- **Sentiment Analysis**: Use OpenAI API to identify sentiment of text as positive/negative
- **Text Extraction**: Extract structured information from unstructured data using JSON schemas

Sentiment Analysis example:

```
curl https://api.openai.com/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
    "model": "gpt-4o-mini",
    "messages": [{ "role": "user", "content": "Write a haiku about programm
  }'
```

Text Extraction with JSON Schema:

```
 curl https://api.openai.com/v1/chat/completions \
-H "Authorization: Bearer $OPENAI_API_KEY" \
-H "Content-Type: application/json" \
-d '{
  "model": "gpt-4o-2024-08-06",
  "messages": [
    { "role": "system", "content": "You are a helpful math tutor. Guide the
```

```
    { "role": "user", "content": "how can I solve 8x + 7 = -23" }
  ],
  "response_format": {
    "type": "json_schema",
    "json_schema": {
      "name": "math_response",
      "strict": true,
      "schema": {
        "type": "object",
        "properties": {
          "steps": {
            "type": "array",
            "items": {
              "type": "object",
              "properties": { "explanation": { "type": "string" }, "output"
              "required": ["explanation", "output"],
              "additionalProperties": false
            }
          },
          "final_answer": { "type": "string" }
        },
        "required": ["steps", "final_answer"],
        "additionalProperties": false
      }
    }
  }
}'
```

Key concepts:

- **Zero-shot, One-shot, Multi-shot Learning**: Different approaches to using LLMs
- **Tokenization**: Impact on LLM input and cost
- **Structured Outputs**: Ensures consistent JSON responses
- **JSON Schema**: Defines expected output structure with validation

Base64 Encoding

- **Purpose**: Convert binary data into ASCII text for transmission through text-only channels
- **How it works**: Takes 3 bytes (24 bits) and converts them into 4 ASCII characters

- **Characters**: A-Z, a-z, 0-9, + and / (padding with `=` to make length multiple of 4)
- **Overhead**: Adds ~33% overhead (every 3 bytes becomes 4 characters)

Python operations:

```
import base64

# Basic encoding/decoding
text = "Hello, World!"
encoded = base64.b64encode(text.encode()).decode()  # SGVsbG8sIFdvcmxkIQ==
decoded = base64.b64decode(encoded).decode()        # Hello, World!

# URL-safe base64
url_safe = base64.urlsafe_b64encode(text.encode()).decode()

# Working with binary files
with open('image.png', 'rb') as f:
    binary_data = f.read()
    image_b64 = base64.b64encode(binary_data).decode()

# Data URI example
data_uri = f"data:image/png;base64,{image_b64}"
```

Common uses:

- JSON: Encoding binary data in JSON payloads
- Email: MIME attachments encoding
- Auth: HTTP Basic Authentication headers
- JWT: Encoding tokens in web authentication
- SSL/TLS: PEM certificate format

Vision Models

- **Purpose**: Use LLMs to interpret images and extract useful information
- **Capabilities**: Detailed textual descriptions, data extraction, object detection

OpenAI Vision API example:

```
curl https://api.openai.com/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
```

```
  -d '{
    "model": "gpt-4o-mini",
    "messages": [
      {
        "role": "user",
        "content": [
          {"type": "text", "text": "What is in this image?"},
          {
            "type": "image_url",
            "image_url": {
              "url": "https://upload.wikimedia.org/wikipedia/commons/3/34/C
              "detail": "low"
            }
          }
        ]
      }
    ]
  }'
```

Base64 image example:

```
 # Download image and convert to base64
IMAGE_BASE64=$(curl -s "https://upload.wikimedia.org/wikipedia/commons/3/34

# Send to OpenAI API
curl https://api.openai.com/v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d @- << EOF
{
  "model": "gpt-4o-mini",
  "messages": [
    {
      "role": "user",
      "content": [
        {"type": "text", "text": "What is in this image?"},
        {
          "type": "image_url",
          "image_url": { "url": "data:image/png;base64,$IMAGE_BASE64" }
        }
      ]
    }
  ]
}
EOF
```

Key features:

- **Detail levels**: `low` (fewer tokens) vs `high` (more detail)
- **Cost management**: Adjust detail settings to balance cost and precision
- **Data extraction**: Convert extracted data to Markdown tables or JSON arrays
- **Model hallucinations**: Address inaccuracies with different prompts

Model selection

- Chat vs embeddings vs vision/speech; pick per task.
- Cost vs speed vs quality; use smaller models for drafts, larger for final.

Exam asks

- Swap base URL and token for proxy usage.
- When to use flex tier and consequences.
- Why structured outputs reduce downstream errors.
- Three prompt tactics that measurably improve outputs and why.

Advanced theory and tricky exam asks

- **Tokenization (BPE)**: Models operate on tokens; longer prompts cost more; formatting (JSON/XML) can add tokens—optimize structure.
- **Decoding controls**: Temperature (randomness) vs top-p (probability mass); higher values increase diversity at the cost of stability.
- **Function calling vs tools**: Structured tool schemas enforce arguments and reduce hallucinations; handle timeouts and retries.
- **Embeddings math**: Cosine similarity ≈ angle; dot product scales with norm —normalize vectors for fair comparison; dimensionality affects recall.
- **Safety/guardrails**: Use instructions + JSON schemas + post-hoc validation; never execute model output blindly.
- **Evals**: Use held-out prompts with exact-match or rubric scoring; log latency, cost, pass@k; benchmark per task, not generic.

What to remember

- Keep prompts concrete: role + context + constraints + examples.
- Normalize embeddings before cosine; store and index with metadata.

- Temperature/top-p trade stability for diversity—lower for tools, higher for ideation.
- Validate JSON outputs; never execute generated code without review.

Embeddings (text and multimodal)

- Purpose: map content to vectors for search, clustering, classification, and RAG.
- Why it matters: enables fast semantic lookup beyond keywords.
- Core concepts:
  - Model choice: small/cheap for search; larger for subtle semantics.
  - Normalization: unit-length for cosine similarity; store metadata (source, chunk id).
  - Chunking: 250–800 tokens; respect sentence/heading boundaries.

Local vs API embeddings:

Feature	Local Models	API
Privacy	High	Dependent on provider
Cost	High setup, low after that	Pay-as-you-go
Scale	Limited by local resources	Easily scales with demand
Quality	Varies by model	Typically high

Local embeddings example:

```
from sentence_transformers import SentenceTransformer
import numpy as np

model = SentenceTransformer('BAAI/bge-base-en-v1.5')

async def embed(text: str) -> list[float]:
    """Get embedding vector for text using local model."""
    return model.encode(text).tolist()

async def get_similarity(text1: str, text2: str) -> float:
    """Calculate cosine similarity between two texts."""
    emb1 = np.array(await embed(text1))
```

```
    emb2 = np.array(await embed(text2))
    return float(np.dot(emb1, emb2) / (np.linalg.norm(emb1) * np.linalg.nor
```

OpenAI embeddings example:

```
import os
import httpx

async def embed(text: str) -> list[float]:
    """Get embedding vector for text using OpenAI's API."""
    async with httpx.AsyncClient() as client:
        response = await client.post(
            "https://api.openai.com/v1/embeddings",
            headers={"Authorization": f"Bearer {os.environ['OPENAI_API_KEY'
            json={"model": "text-embedding-3-small", "input": text}
        )
        return response.json()["data"][0]["embedding"]
```

- Pitfalls: mixing different embedding models between index and query; chunks too small lose context, too big blur relevance.
- Checklist: pick model; define chunking; normalize/store vectors with metadata; verify retrieval on a gold set.

Multimodal Embeddings

- **Purpose**: Map text and images into the same vector space for cross-modal comparison
- **Applications**: Cross-modal search, content recommendation, clustering & retrieval, anomaly detection

Providers and setup:

- **Nomic Atlas**: Sign up at atlas.nomic.ai, get API key from Settings
- **Jina AI**: Visit jina.ai/embeddings/, 1 million free tokens
- **Google Vertex AI**: Sign up for Google Cloud free tier, create API key

Nomic Atlas example:

```
# Text embeddings
curl -X POST "https://api-atlas.nomic.ai/v1/embedding/text" \
  -H "Authorization: Bearer $NOMIC_API_KEY" \
```

```
  -H "Content-Type: application/json" \
  -d '{
        "model": "nomic-embed-text-v1.5",
        "task_type": "search_document",
        "texts": ["A cute cat", "A cardboard box"]
      }'

# Image embeddings
curl -X POST "https://api-atlas.nomic.ai/v1/embedding/image" \
  -H "Authorization: Bearer $NOMIC_API_KEY" \
  -F "model=nomic-embed-vision-v1.5" \
  -F "images=@cat.jpg" \
  -F "images=@box.png"
```

Topic Modeling

- **Purpose**: Use text embeddings to find text similarity and create topics automatically
- **Applications**: Document clustering, content organization, trend analysis

Key concepts:

- **Embeddings**: How LLMs convert text into numerical representations
- **Similarity Measurement**: Understanding how similar embeddings indicate similar meanings
- **Cosine Similarity**: Calculating similarity between embeddings for reliable measures
- **Embedding Visualization**: Using tools like Tensorflow Projector to visualize embedding spaces

Tools and resources:

- [Tensorflow projector](#) for visualization
- [Massive text embedding leaderboard (MTEB)](#)
- [Embeddings similarity threshold](#)
- [Clustering on scikit-learn](#)

Vector databases (ANN search)

- Purpose: retrieve nearest neighbors quickly from large corpora.
- Why: brute-force search is O(n); ANN gives sublinear with good recall.

- Core concepts: namespaces, metadata filtering, HNSW/IVF indexes, recall vs latency tuning.
- Workflow: metadata filter → ANN search (k) → optional re-rank.
- Local-first: start with a local index for prototypes; move to managed only when scale/ops require.
- Pitfalls: forgetting metadata filters; poor index parameters; mixing spaces (cosine vs dot) incorrectly.
- Checklist: choose distance metric; set index params; add metadata filters; test recall vs latency.

Retrieval-Augmented Generation (RAG)

- Purpose: ground answers in your corpus.
- Pipeline: chunk/embed → retrieve top-k → optionally re-rank → compose answer with citations.
- Prompts: instruct citation format and refusal when no evidence.

```
Answer using only retrieved passages. Cite [doc_id:page] after each claim.
```

- Hybrid retrieval: combine keyword (BM25) + vector for recall; re-rank with cross-encoders for precision.
- Pitfalls: hallucinated citations; retrieval mismatch; stale indexes.
- Checklist: chunking policy; retriever config; re-ranker optional; eval over a labeled Q/A set.

Function calling (tool use)

- Purpose: let models call deterministic functions via structured args.
- Core concepts: JSON schema, validation, idempotency, timeouts/retries, auth separation.
- Example schema:

```
{
  "name": "search_flights",
  "parameters": {
    "type": "object",
    "properties": {
      "from": {"type": "string", "pattern": "^[A-Z]{3}$"},
```

```
      "to": {"type": "string", "pattern": "^[A-Z]{3}$"},
      "date": {"type": "string", "format": "date"}
    },
    "required": ["from","to","date"]
  }
}
```

- Pitfalls: overbroad schemas; non-idempotent side effects; unbounded retries.
- Checklist: strict schema; server-side validation; timeouts; retry with backoff; log tool I/O.

Agents (decision loops)

- Purpose: orchestrate multi-step tasks using tools.
- Keep simple: minimal toolset; explicit stop conditions; self-check before act.
- Safety: max depth; budget caps; sandbox side effects.
- Logging: store state transitions and tool I/O for audit.
- Pitfalls: loops, tool hallucinations, prompt drift.
- Checklist: define goal/stop; limit tools; add self-critique; enforce depth/timeout.

Evals (measuring quality)

- Purpose: quantify task performance and regressions.
- Design: small, representative task set; include edge cases.
- Scoring: exact match where possible; rubric when open-ended.
- Ops: track latency, cost, pass@k; analyze failure clusters.
- Example rubric:

```
Summarize in 3 bullets (80–120 words). Score 0–2 each: coverage, faithfuln
```

- Checklist: dataset/versioning; scorer; dashboards; regression alerts.

Multimodal (images, audio, vision)

- Purpose: work across text, images, audio/video.
- Specify: task (caption, OCR, detection), size constraints, output JSON for boxes/labels.

- Ops: batch/caching for TTS/STT and image pipelines; hash content to dedupe.
- Pitfalls: oversized payloads; ambiguous task prompts; inconsistent formats.
- Checklist: define modality; compress/resize; specify output schema; cache results.

Realtime and streaming

- Purpose: responsive UIs and long outputs.
- UX: incremental rendering, cancel/retry, partial copy.
- Server: chunked responses, back-pressure, timeouts.
- Pitfalls: unbounded streams; stalled connections; token overruns.
- Checklist: streaming protocol, token budgets, idle timeouts, retry policy.

Website/Video scraping with LLMs

- Approach: render JS when needed, prefer selector-based extraction, store raw HTML/frames.
- Ethics: rate-limit, respect terms, attribute sources.
- Pitfalls: relying on free-text extraction alone; unstable selectors; legal pitfalls.
- Checklist: stable locators; storage for raw data; throttle and identify client.

Local models and hosting

- Purpose: privacy, cost control, offline.
- Ops: pin model versions, monitor memory/throughput, benchmark latency/quality.
- Hardware: choose CPU vs GPU vs quantized variants based on budget and concurrency.
- Pitfalls: memory fragmentation; model drift; mismatched quantization.
- Checklist: versioning; resource monitors; perf tests; fallbacks.

Vision models

- Purpose: perception tasks; combine with text prompts.
- Specify: precise instructions and expected structured outputs.

- Preprocess: compress/standardize images; control resolution to fit budgets.
- Pitfalls: non-deterministic bounding boxes, color space issues, EXIF rotations.
- Checklist: task definition; output schema; image preprocessing pipeline; QA on samples.

LLM Image Generation

- **Purpose**: Generate and edit images using LLMs like Gemini 2.0 Flash and GPT Image 1
- **Capabilities**: Text-to-image generation, image editing, style control

Gemini 2.0 Flash Experimental:

```
curl "https://generativelanguage.googleapis.com/v1beta/models/gemini-2.0-f
  -H "Content-Type: application/json" \
  -X POST \
  -d '{
    "contents": [{ "parts": [{ "text": "A serene landscape of rolling hills
    "generationConfig": { "responseModalities": ["TEXT", "IMAGE"] }
  }' | jq -r '.candidates[].content.parts[] | select(.inlineData) | .inline
```

OpenAI GPT Image 1:

```
curl 'https://api.openai.com/v1/images/generations' \
  -H 'Content-Type: application/json' \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -d '{
    "model": "gpt-image-1",
    "prompt": "A whimsical illustration of a cat playing chess",
    "n": 1,
    "size": "1024x1024"
  }' > image.png
```

Generation options:

- `temperature` (0.0–2.0): Controls randomness
- `topP` (0.0–1.0): Nucleus sampling threshold
- `maxOutputTokens`: Max tokens for text parts
- `size`: Image dimensions (256x256, 512x512, 1024x1024)

LLM Speech Generation

- **Purpose**: Convert text to natural-sounding speech using TTS models
- **Providers**: OpenAI TTS-1, Google Gemini Speech Studio

OpenAI TTS-1:

```
curl https://api.openai.com/v1/audio/speech \
  -H "Authorization: Bearer $OPENAI_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "tts-1",
    "input": "Hello! This is a test of the OpenAI text to speech API.",
    "voice": "alloy"
  }' --output speech.mp3
```

Google Gemini Speech Studio:

```
curl -X POST "https://texttospeech.googleapis.com/v1/text:synthesize?key=$(
  -H "Content-Type: application/json" \
  -d '{
    "input": { "text": "Hello, welcome to Gemini Speech Studio!" },
    "voice": { "languageCode": "en-US", "name": "en-US-Neural2-A" },
    "audioConfig": { "audioEncoding": "MP3" }
  }' | jq -r .audioContent | base64 --decode > gemini-speech.mp3
```

Voice options:

- **OpenAI**: `alloy`, `echo`, `fable`, `onyx`, `nova`, `shimmer`
- **Google**: Various neural voices with different languages and genders

SSML support:

```
<speak>Hello <break time="1s"/> This text has a pause and <emphasis level=
```

LLM Evaluations with PromptFoo

- **Purpose**: Test-drive prompts and models with automated, reliable evaluations
- **Features**: Multi-provider support, built-in assertions, CI/CD integration

Setup:

```yaml
# promptfooconfig.yaml
prompts:
  - |
    Summarize this text: "{{text}}"
  - |
    Please write a concise summary of: "{{text}}"

providers:
  - openai:gpt-3.5-turbo
  - openai:gpt-4

tests:
  - name: summary_test
    vars:
      text: "PromptFoo is an open-source CLI and library for evaluating and
    assertions:
      - contains-all:
          values:
            - "open-source"
            - "LLMs"
      - llm-rubric:
          instruction: |
            Score the summary from 1 to 5 for:
            - relevance: captures the main info?
            - clarity: wording is clear and concise?
          schema:
            type: object
            properties:
              relevance:
                type: number
                minimum: 1
                maximum: 5
              clarity:
                type: number
                minimum: 1
                maximum: 5
            required: [relevance, clarity]
            additionalProperties: false
```

Usage:

```
# Execute all tests
npx -y promptfoo eval -c promptfooconfig.yaml

# Launch interactive results viewer
```

```
npx -y promptfoo view -p 8080

# Disable cache for fresh results
echo y | promptfoo eval --no-cache -c promptfooconfig.yaml
```

Key features:

- **Developer-first**: Fast CLI with live reload & caching
- **Multi-provider**: Works with OpenAI, Anthropic, HuggingFace, Ollama & more
- **Assertions**: Built-in (`contains`, `equals`) & model-graded (`llm-rubric`)
- **CI/CD**: Integrate evals into pipelines for regression safety