BASWE.AI

15 LLM Tricks for Applied AI Engineers

Production-grade prompt engineering, system design, and cost optimization. Each trick is immediately deployable.

CONTENTS

System prompt is a contract

Role + constraints in system. Dynamic input in user. Never mix them.

WHY THIS MATTERS

The system prompt is evaluated once and cached. The user message is dynamic. When you pollute your system prompt with runtime data, you lose caching benefits, make your prompt harder to test, and give the model conflicting instructions. Think of it like a class definition vs an instance call.

HOW TO APPLY IT

-> Put persona, tone, output format, and hard constraints in system.

-> Put the actual user request, document content, or variables in user.

-> Never interpolate dynamic data into the system prompt — that's what the user turn is for.

-> Test your system prompt in isolation with a variety of user messages.

WATCH OUT:

If your system prompt contains f-strings or runtime variables, you've already broken this rule. Refactor immediately.

EXAMPLE

```
system = """
You are a senior code reviewer.
Rules:
- Flag security issues only
- One JSON object per issue
- No prose, no explanations
"""

user = f"Review this: {code_snippet}"

# system = stable, cacheable contract
# user   = dynamic per-request input
```

Separating concerns = consistent, testable, cheaper outputs.

One prompt, five tasks = garbage

Chain outputs. Each prompt does one job. Pass the result forward.

WHY THIS MATTERS

LLMs are autoregressive — they generate one token at a time and can lose track of multiple competing objectives. When you ask a single prompt to summarize AND translate AND extract AND reformat, you're betting that the model holds all four goals equally in mind. It won't. Quality degrades with each additional task.

HOW TO APPLY IT

-> Decompose your pipeline into single-responsibility prompts.

-> Feed the output of step N as the input to step N+1.

-> Each step is independently testable and replaceable.

-> Use a smaller, cheaper model for simple steps — save the frontier model for hard reasoning.

WATCH OUT:

Chaining adds latency. If speed matters, identify which steps can run in parallel and batch them.

EXAMPLE

```
# bad: one prompt doing too much
prompt = "summarize, translate to Spanish,
  extract named entities, score sentiment,
  and return everything as JSON"

# good: chained single-responsibility steps
summary  = llm(summarize_prompt,  doc)
translated = llm(translate_prompt, summary)
entities = llm(extract_prompt,    translated)
output   = llm(format_json_prompt, entities)
```

Smaller prompts = fewer hallucinations + isolated failure points.

Stop parsing. Force structure.

Define a JSON schema upfront. The model returns exactly what your app expects.

WHY THIS MATTERS

Free-text LLM output fed into a regex or string parser is a production time bomb. The model might say 'the sentiment is positive' one call and 'Positive.' the next. Your parser breaks, your pipeline fails silently, and you spend hours debugging a formatting issue instead of improving your product.

HOW TO APPLY IT

-> Use response_format with json_schema (OpenAI) or tool_use (Anthropic) to enforce structure.

-> Define required fields, types, and enums in your schema.

-> Validate the parsed response against your schema before passing it downstream.

-> Use Pydantic models to auto-generate schemas from your data classes.

WATCH OUT:
Structured output adds a small quality cost on very open-ended creative tasks. For extraction, classification, and data tasks — always use it.

EXAMPLE

```
from pydantic import BaseModel

class Extraction(BaseModel):
    entities: list[str]
    sentiment: str  # "positive"|"negative"|"neutral"
    confidence: float

response = client.beta.chat.completions
  .parse(
    model="gpt-4o",
    messages=[...],
    response_format=Extraction
  )

result = response.choices[0].message.parsed
# result.sentiment -> guaranteed string, no parsing
```

Schema-enforced output is production-ready from call one.

Random outputs break evals.

Lock your seed during testing. Same input = same output = trustworthy benchmarks.

WHY THIS MATTERS

Without a fixed seed, two identical prompts can return different outputs. This makes it impossible to know if a prompt change improved quality or if you just got lucky with sampling. You need reproducibility to run meaningful A/B tests, regression evals, and debugging sessions.

HOW TO APPLY IT

-> Set seed=42 and temperature=0 during all development and evaluation runs.

-> Store the seed alongside every logged call so you can replay it exactly.

-> Once you're satisfied with prompt quality, remove the seed for production variety.

-> Note: seed is best-effort on some providers — verify determinism with your specific model.

WATCH OUT:
temperature=0 alone is not sufficient for full determinism — some providers use non-deterministic sampling even at zero. Always combine with seed.

EXAMPLE

```
# development / evaluation
response = client.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": prompt}],
    temperature=0,   # no randomness
    seed=42          # deterministic sampling
)

# log it for replay
log({"seed": 42, "prompt": prompt,
     "output": response.choices[0].message.content})

# production: remove seed, set temperature > 0
```

You cannot improve what you cannot reproduce.

Few-shot > zero-shot for edge cases

When the task has nuance, show 2-3 examples. Don't just describe the behavior.

WHY THIS MATTERS

Instructions describe intent. Examples demonstrate execution. For tasks with subtle output requirements — tone calibration, edge-case classification, domain-specific formatting — the model learns faster from seeing the pattern than from reading about it. Think of examples as in-context fine-tuning.

HOW TO APPLY IT

-> Pick 2-3 examples that cover your most common edge cases, not just the happy path.
-> Format examples identically to your real inputs — the pattern matters.
-> Include at least one negative or tricky example to bound the output space.
-> Order matters: put the most representative example last, closest to the actual input.

WATCH OUT:

More examples = more tokens = more cost. 3-5 examples is usually the sweet spot. Beyond that, consider fine-tuning.

EXAMPLE

```
prompt = """
Classify support ticket urgency.

Ticket: "production server is down"
Urgency: critical

Ticket: "update my billing email"
Urgency: low

Ticket: "charts not loading on dashboard"
Urgency: medium

Ticket: "{user_ticket}"
Urgency:"""

# the model continues the pattern
# no instructions needed — it sees the format
```

Examples teach faster than instructions. Always show the pattern.

Use roles to unlock behavior

Telling the model WHO it is changes how it reasons. Be specific — not generic.

WHY THIS MATTERS

The model has been trained on output from thousands of different types of experts. When you specify a role precisely, you activate the relevant knowledge cluster. A vague role like 'helpful assistant' gives you averaged, generic output. A specific role like 'staff engineer at a fintech startup who prioritizes security' narrows the distribution significantly.

HOW TO APPLY IT

-> Include seniority level, domain, and company context in the role.

-> Add behavioral constraints: 'you are direct, you don't sugarcoat, you flag risks first'.

-> Specify the audience: 'you are explaining this to a junior engineer' changes depth and vocabulary.

-> Combine role with goal: 'You are a technical writer whose job is to make dense API docs readable.'

WATCH OUT:

Don't confuse role-setting with jailbreaking. Roles shape reasoning style and domain depth — they don't bypass safety constraints.

EXAMPLE

```
# weak: generic role
system = "You are a helpful assistant."

# strong: specific role with constraints
system = """
You are a staff-level backend engineer
at a Series B fintech company.

You:
- Write production Python, not tutorial code
- Flag security risks before anything else
- Prefer explicit error handling over try/except
- Never use mutable default arguments
- Return only code, no explanations
"""
```

Vague roles get vague answers. Precision unlocks expertise.

Retrieval context placement matters

Put retrieved chunks at the TOP of the prompt. LLMs attend better to early context.

WHY THIS MATTERS

Research on 'lost in the middle' shows that LLMs recall information placed at the beginning and end of a long context window much better than information buried in the middle. In RAG systems, most engineers place retrieved context after the instruction — exactly where it's most likely to be ignored.

HOW TO APPLY IT

-> Structure your RAG prompt: CONTEXT first, then INSTRUCTIONS, then QUERY.

-> If you have multiple chunks, put the highest-relevance chunk first.

-> For very long contexts, also repeat the key fact at the end as a reinforcement.

-> Limit context to what's relevant — don't pad with low-similarity chunks to fill the window.

WATCH OUT:

More context is not always better. Irrelevant retrieved chunks add noise and increase hallucination risk. Set a similarity threshold — don't retrieve below it.

EXAMPLE

```
# bad: context buried after instructions
prompt = f"""
Answer the user question accurately.
Be concise. Use bullet points.

Question: {query}

Context: {retrieved_chunks}  <- buried!
"""

# good: context first
prompt = f"""
Context:
{retrieved_chunks}

Instruction: Answer using only the context above.
Question: {query}
"""
```

Context placement directly impacts answer accuracy in RAG pipelines.

Think step-by-step for hard tasks

Force chain-of-thought. Structured reasoning dramatically improves accuracy.

WHY THIS MATTERS

LLMs generate tokens sequentially — they don't think ahead. When you ask for an answer directly, the model gives you its best first guess. When you force step-by-step reasoning, the model builds toward the answer using its own intermediate outputs as context. This is especially critical for math, logic, and multi-condition decisions.

HOW TO APPLY IT

-> Explicitly define the reasoning steps in your prompt — don't just say 'think step by step'.

-> Ask the model to output its reasoning before its final answer.

-> For complex decisions, use a 'scratchpad' pattern: reasoning in one field, answer in another.

-> In JSON output, add a 'reasoning' key before the 'answer' key — order matters for token generation.

WATCH OUT:

Chain-of-thought adds output tokens and therefore cost. For simple classification tasks, it's overkill. Reserve it for multi-step reasoning.

EXAMPLE

```
# without CoT: model guesses directly
prompt = "Is this code vulnerable? Yes or No."

# with CoT: model reasons to the answer
prompt = """
Analyze this code for vulnerabilities.

Step 1: List all user-controlled inputs.
Step 2: Trace each input to its output.
Step 3: Check each path for sanitization.
Step 4: State your verdict with evidence.

Code: {code}
"""

# structured reasoning = auditable output
```

Chain-of-thought is free compute. Always use it for hard reasoning tasks.

Stream for user-facing apps

Never block your UI waiting for full LLM completion. Stream from the first token.

WHY THIS MATTERS

A GPT-4o call generating 500 tokens takes 5-8 seconds to complete. Showing a blank screen for 7 seconds kills perceived quality. With streaming, the first tokens appear in under 500ms. Users tolerate slow typing — they won't tolerate frozen interfaces. Streaming is a UX requirement, not an optimization.

HOW TO APPLY IT

-> Set stream=True on any user-facing LLM call.

-> Pipe each chunk to the frontend via SSE (Server-Sent Events) or WebSockets.

-> Handle stream interruption gracefully — users close tabs mid-generation.

-> For structured output tasks, buffer the full stream before parsing JSON — don't parse mid-stream.

WATCH OUT:
Streaming complicates error handling. The HTTP 200 has already been sent before you know if the generation will fail. Implement a fallback for truncated or malformed streams.

EXAMPLE

```
stream = client.chat.completions.create(
    model="gpt-4o",
    messages=[{"role": "user", "content": prompt}],
    stream=True
)

for chunk in stream:
    delta = chunk.choices[0].delta.content
    if delta:
        # send to frontend via SSE
        yield f"data: {delta}\n\n"

# FastAPI + SSE example:
# return StreamingResponse(stream_llm(),
#   media_type="text/event-stream")
```

Users tolerate slow. They will not tolerate frozen.

Eval before you ship

Build a golden dataset before touching production. Vibe-checking is not evaluation.

WHY THIS MATTERS

Prompt engineering without evals is guessing. You change a prompt, it 'feels better' on 2 examples, and you ship it. Then it silently regresses on 15% of real inputs you never tested. A golden dataset — even 20 input/output pairs — gives you a regression number. That number is the difference between engineering and hoping.

HOW TO APPLY IT

-> Build a golden set of 20-50 real input/expected output pairs before writing any prompt.

-> Run your eval suite on every prompt change before merging.

-> Track scores over time — a drop from 94% to 88% is a signal, even if it 'feels fine'.

-> Use LLM-as-judge for subjective quality (pass the output + rubric to a grader model).

WATCH OUT:
Golden datasets go stale. Refresh them when your input distribution changes — new user segments, new features, seasonal content shifts.

EXAMPLE

```
evals = [
  {"input": "classify: server is down",
   "expected": "critical"},
  {"input": "classify: update my email",
   "expected": "low"},
  # ... 18 more
]

def run_eval(prompt_fn):
    results = [
        prompt_fn(e["input"]) == e["expected"]
        for e in evals
    ]
    score = sum(results) / len(results)
    print(f"Accuracy: {score:.1%}")  # e.g. 92.0%
    return score
```

Ship with a number. Not a feeling.

Negative prompting cuts hallucination

Explicitly tell the model what NOT to do. Silence on edge cases = confident wrong answers.

WHY THIS MATTERS

LLMs are trained to be helpful, which means they'll try to answer even when they shouldn't. Without a fallback instruction, the model will confidently fabricate an answer rather than admit it doesn't know. Negative constraints create the permission structure for the model to say 'I don't know' — and they need that permission explicitly.

HOW TO APPLY IT

-> Always include a 'if you don't know, say X' clause in system prompts for RAG and Q&A tasks.

-> Use 'Do NOT' language for hard rules — softer phrasing like 'try to avoid' is often ignored.

-> Test your negative constraints by asking questions the model definitely can't answer from context.

-> Combine negative prompting with source attribution: 'cite the passage you drew from or do not answer'.

WATCH OUT:
Over-constraining leads to refusal on valid questions. Calibrate by testing your constraint set on a mix of answerable and unanswerable inputs.

EXAMPLE

```
system = """
You are a support agent for AcmeCorp.
Answer using ONLY the provided knowledge base.

If the answer is not in the knowledge base:
  Respond exactly: "I don't have that information.
  Please contact support@acmecorp.com"

Do NOT guess.
Do NOT use information from your training data.
Do NOT apologize — just give the fallback response.
"""
```

Explicit constraints prevent confident wrong answers at scale.

Tool calling > string parsing

Define tools so the model triggers actions directly. Parsing free-text intent is brittle.

WHY THIS MATTERS

Early LLM integrations parsed natural language output to determine what action to take: 'if response contains SEARCH, run a search'. This breaks constantly. Tool calling (function calling) inverts the pattern — you define a typed function signature, and the model decides when and how to call it. The model handles intent detection; your code handles execution.

HOW TO APPLY IT

-> Define one tool per action — don't create a mega-tool with 20 optional parameters.

-> Use clear, descriptive tool names and parameter descriptions — the model reads these.

-> Always handle the case where the model doesn't call any tool (direct answer path).

-> For agentic systems, expose a small, well-defined tool surface. Fewer tools = better decisions.

WATCH OUT:
Tool descriptions are part of your prompt and count toward your token budget. Keep them concise but complete — the model uses them to decide when to call.

EXAMPLE

```
tools = [{
  "type": "function",
  "function": {
    "name": "search_knowledge_base",
    "description": "Search internal docs for an answer.",
    "parameters": {
      "type": "object",
      "properties": {
        "query": {"type": "string",
                  "description": "search terms"},
        "limit": {"type": "integer",
                  "description": "max results, default 5"}
      },
      "required": ["query"]
    }
  }
}]
```

Tool calling is the correct abstraction for agentic AI systems.

Cache aggressively on repeat prompts

Static system prompts should always be cached. Pay only for the dynamic part.

WHY THIS MATTERS

Prompt caching lets providers reuse the KV cache from a previous call if the prompt prefix matches. For apps with a large, stable system prompt — a 2,000-token RAG instruction set, a 500-token persona, a full document being repeatedly queried — caching cuts input token cost by up to 90% and drops latency by 30-50% on cache hits.

HOW TO APPLY IT

-> On Anthropic: add cache_control: {type: ephemeral} to your system prompt content block.
-> On OpenAI: caching is automatic for prompts over 1,024 tokens — no configuration needed.
-> Structure your prompt so the static part comes FIRST — the cache only applies to the prefix.
-> Monitor cache hit rates in your provider dashboard — a low rate means your prompt structure is wrong.

WATCH OUT:
Ephemeral caches expire after ~5 minutes of inactivity. For infrequent use cases, you may not see cache benefits. Best ROI on high-volume, low-variation workloads.

EXAMPLE

```
# Anthropic: explicit cache control
response = client.messages.create(
  model="claude-sonnet-4-5",
  system=[
    {
      "type": "text",
      "text": large_system_prompt,  # 2000+ tokens
      "cache_control": {"type": "ephemeral"}
    }
  ],
  messages=[{"role":"user","content": query}]
)

# check cache usage
print(response.usage.cache_read_input_tokens)
# cache hit = ~90% cost reduction on prefix
```

Cache the static. Pay only for the dynamic.

Log inputs + outputs in production

You cannot debug what you didn't log. Every LLM call needs an observable trace.

WHY THIS MATTERS

LLM failures are probabilistic and hard to reproduce. A user reports a bad output — without logs, you have no prompt, no model version, no temperature setting, no context. You can't reproduce the failure, which means you can't fix it. Structured logging of every LLM call is the minimum viable observability layer for any production AI system.

HOW TO APPLY IT

-> Log: prompt, model, temperature, seed, response, latency, token counts, and timestamp.

-> Assign a trace_id to each call so you can correlate multi-step pipelines.

-> Store logs in a queryable store (Postgres, BigQuery) — not just application logs.

-> Flag and review low-confidence or fallback responses automatically.

WATCH OUT:
Logs contain user data. Implement PII scrubbing before writing to your log store. Define a retention policy — raw LLM logs grow fast.

EXAMPLE

```
import time, uuid

def llm_call(prompt: str, **kwargs) -> str:
    trace_id = str(uuid.uuid4())
    t0 = time.time()

    response = client.chat.completions.create(
        messages=[{"role":"user","content":prompt}],
        **kwargs
    )

    output = response.choices[0].message.content
    db.log({
        "trace_id":  trace_id,
        "prompt":    prompt,
        "output":    output,
        "model":     kwargs.get("model"),
        "latency_ms": int((time.time()-t0)*1000),
        "tokens":    response.usage.total_tokens
    })
    return output
```

Observability is not optional. Log from day one or debug forever.

Model routing cuts cost by 60%+

Not every call needs GPT-4o. Route by complexity. Reserve frontier models for hard reasoning.

WHY THIS MATTERS

GPT-4o costs 17x more per token than GPT-4o-mini. If 70% of your calls are simple extraction, classification, or formatting tasks — tasks a smaller model handles just as well — you're burning money. Model routing is the single highest-ROI optimization available to a production AI engineer after architecture is set.

HOW TO APPLY IT

-> Classify task complexity before routing — use a fast, cheap model to classify.
-> Define clear boundaries: simple = extraction/format/classify, medium = summarization/QA, hard = reasoning/code.
-> Track accuracy by tier — if your 'simple' tier degrades quality, adjust the boundary.
-> Use the frontier model as a fallback, not a default.

WATCH OUT:
Routing adds one extra LLM call (the classifier). Make sure the cost savings from routing outweigh the classifier cost. At scale they always do — verify at your volume.

EXAMPLE

```
MODELS = {
    "simple": "gpt-4o-mini",   # $0.15/M tokens
    "medium": "gpt-4o",        # $2.50/M tokens
    "hard":   "o3",            # $10.00/M tokens
}

def route_and_call(task: str) -> str:
    # cheap classifier decides tier
    tier = classify_complexity(task)  # simple|medium|hard
    model = MODELS[tier]

    response = client.chat.completions.create(
        model=model,
        messages=[{"role":"user","content":task}]
    )
    log({"task": task, "tier": tier, "model": model})
    return response.choices[0].message.content
```

Intelligence tiers exist for a reason. Route intelligently — save the frontier for what needs it.