AI & LLMS

Building LLM-Powered Applications

Architecture, evaluation and guardrails for production

Houssam Kodad

PDF · DATAFORGE BOOKS

CONTENTS

Table of Contents

CHAPTER 01

From Demo to Dependable Product

A working demo with a language model takes an afternoon. A product people can rely on takes engineering. The gap between the two is where this entire book lives: grounding the model in your own data, measuring whether it is actually correct, controlling cost and latency under real traffic, and putting guardrails in place before users discover the edge cases for you. The model is the easy part; the system around it is the product.

This chapter frames that gap honestly. We look at what genuinely changes when you move from a prototype to production, sketch a reference architecture for an LLM application, and argue for abstracting the model behind an interface rather than welding your product to one provider's API. These are the structural decisions that determine whether your application survives contact with real users and a changing model landscape.

The approach throughout is model-agnostic and grounded in patterns that outlast any single API. Providers and model names change every few months; the engineering of retrieval, evaluation, guardrails and cost control does not. We deliberately build on the durable parts, so that what you learn here remains useful long after today's favourite model has been superseded.

What Changes at Production Scale

In a demo, you control the input, run it once, and judge the output by eye. In production, real users send inputs you never imagined, the same prompt runs millions of times, and "looks good to me" is replaced by hard questions: is it correct, how do you know, what does it cost per request, and what happens when it fails at three in the morning. Each of these questions forces engineering you happily skipped in the demo.

Correctness needs evaluation you can run automatically. Cost needs caching and model routing. Failure needs guardrails and fallbacks. Variability needs observability so you can see what the model actually did. None of this is exotic; it is the same operational rigour any serious system demands. The novelty is only that a non-deterministic component now sits at the centre, and the rest of the book is about engineering responsibly around that fact.

A Reference Architecture

A production LLM application is a pipeline, not a single call, and naming its stages early gives you places to add reliability without rewriting everything. A request arrives and is validated. Relevant

context is retrieved from your data. A prompt is assembled within a token budget. The model is called, possibly invoking tools. The output is validated and filtered. And the whole interaction is traced so you can evaluate and debug it later.

Each stage is a seam where you will later insert a capability — a guardrail here, a cache there, an evaluation hook around the whole thing. Treating the application as this explicit pipeline, rather than a single function that calls a model, is what makes it possible to harden one part at a time. The book follows this architecture chapter by chapter, treating evaluation and observability as first-class concerns rather than afterthoughts.

```
def handle(request):
    if not valid(request):                 # 1. validate input
        return reject(request)
    context = retrieve(request.query)       # 2. ground in your data
    prompt  = assemble(request, context, budget=6000)  # 3. fit the window
    result  = model.complete(prompt, tools=TOOLS)      # 4. call (with tools)
    safe    = guardrails(result)            # 5. validate + filter output
    trace(request, context, prompt, result) # 6. observe everything
    return safe
```

How LLMs Behave and Fail

To engineer around a component you must understand its failure modes, and language models fail in characteristic ways. They have a fixed context window, so there is a hard limit on how much they can attend to at once. They are confidently wrong — hallucination is not a bug to be patched but an inherent property to be managed. And they are non-deterministic, so the same input can yield different outputs, which complicates both testing and debugging.

None of these are reasons to avoid LLMs; they are design constraints to plan around. Hallucination is mitigated by grounding the model in retrieved facts and asking it to abstain when unsure. Context limits are managed by careful selection of what to include. Non-determinism is tamed by lowering temperature and constraining outputs. Knowing the failure modes turns vague anxiety into a concrete checklist of things to handle.

Prompt and Context Design

The prompt is the interface to the model, and designing it is closer to software design than to clever wording. A production prompt has structure: a system message that sets the role and rules, the user's request, and carefully chosen context. The discipline is to be explicit about what you want, what to avoid, and crucially what format the answer must take — because a parseable answer is one your

code can rely on.

Context-window budgeting is the constraint that shapes everything. You have a fixed token budget, and you must decide what earns a place in it: which retrieved passages, how many examples, how much history. Spend it badly — stuffing in marginally relevant text — and you dilute the signal and pay for tokens that hurt rather than help. We treat the context window as a scarce resource to be allocated deliberately.

Structured Outputs and Tool Use

An LLM that returns free-form prose is hard to build on; an LLM that returns structured, validated data is a reliable component. By instructing the model to respond in a defined schema and validating what comes back, you turn an unpredictable text generator into something you can wire into the rest of your system with confidence. This single technique unlocks most practical applications.

Tool use — letting the model call functions you define, such as a search, a calculation, or a database lookup — extends the model beyond its training data and its arithmetic weaknesses. Done well, it lets the model decide when to fetch a fact or perform an action, with your code firmly in control of what is actually allowed. We cover both structured outputs and tool calling in depth, because together they are what make LLM applications do real work rather than just talk.

```
# Constrain the output so failures are detectable, not silent.
schema = {
    "type": "object",
    "properties": {
        "category": {"enum": ["billing", "technical", "account"]},
        "urgent":   {"type": "boolean"},
    },
    "required": ["category", "urgent"],
}
result = model.complete(prompt, response_format=schema)
ticket = validate(json.loads(result), schema)   # trust nothing unvalidated
```

Evaluation You Can Trust

The question "did that change make the system better or worse?" has no answer without evaluation, and eyeballing a few examples is not evaluation. You need a labelled set of representative inputs with known good outputs, and an automated way to score the system against them, so that every prompt tweak or model swap can be judged on evidence rather than vibes.

Using an LLM to judge another LLM's output is a powerful technique and a treacherous one, prone to its own biases and inconsistencies. We treat evaluation as the serious engineering problem it is — building reliable eval sets, combining automated and human judgement, and guarding against the ways an LLM judge can mislead. Without trustworthy evaluation, you are flying blind, and most of the failures in production LLM systems trace back to its absence.

Guardrails, Latency and Cost

Before real users arrive, you need guardrails: input filtering against abuse and prompt injection, output filtering against unsafe or off-policy responses, and careful handling of personal data. These are not optional polish; they are what stands between your application and the inevitable user who tries to make it misbehave. We treat safety as a designed-in layer, not a content filter bolted on at the end.

Finally, production means living within latency and cost budgets. Streaming responses improves perceived speed; caching avoids paying for the same answer twice; routing simple requests to small, cheap models and hard ones to large models controls spend without sacrificing quality. The chapters ahead build out each of these — retrieval, evaluation, guardrails, cost control — until you can ship LLM features that survive real users, real traffic, and a real budget.

This is a free sample

You've reached the end of the sample chapter.

Get the complete book — every chapter, fully worked — at **dataforgebooks.com**.

FULL EDITION · 288 PAGES · PDF

Read the full title at **dataforgebooks.com**

Questions? support@dataforgebooks.com