


Best Practices for Developing & Deploying LLM Apps in The Real World

You've built a powerful large language model (LLM) application — one that can answer complex questions, generate compelling text, and even reason through tough problems. Now comes the real test: getting it out of the lab and into the hands of users.

Deploying an LLM in production isn't just about having a great model. It's about building a system that's fast, reliable, cost-efficient, and scalable.

 Think of this like launching a spacecraft: the model is your engine, but the architecture is your guidance system, fuel tank, and parachute. Without a solid plan, even the most advanced model can crash.

In this chapter, we'll walk through the full lifecycle of deploying a large language model — from understanding business needs to optimizing performance, managing costs, and ensuring system stability.

The story so far...

You're now ready to take **TaskFriend** to a larger audience - and start offering it to the public. However, before you do, you want to make sure that you've covered all your bases and ensure the system is **fast, reliable, cost-efficient, and scalable in production** — just like launching a spacecraft where the model is the engine, but the architecture is the guidance system, fuel tank, and parachute.

This means validating everything from business requirements and model selection to performance optimization, user experience, and system resilience.

Goals

- Learn how to translate & balance business needs into clear requirements.
- Understand the techniques used to optimize LLM application performance from both application and user perspectives.
- Understand the concerns behind ensuring application reliability & resilience.

Intializing the environment

Setting up the API key

Before we start work on in any notebook, we'll need to load the [API key for Model Studio](#). This ensures that we can call APIs of Qwen models we'll be using throughout this course.

If you're unsure about how to find your Model Studio API key, refer to the [00-2 Setting Up Model Studio for LMP-C01.ipynb](#) file.

```
# Load Model Studio API key
import os
from config.load_key import load_key
load_key(
    confirmation=False
)
```

```
from openai import OpenAI
from functions.llm_utils import (
    get_qwen_stream_response,
    get_hardcoded_response,
    get_precomputed_response,
    benchmark_responses
)
import time

client = OpenAI(
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url="https://dashscope-intl.aliyuncs.com/compatible-mode/v1",
)
```

Translating Business Requirements into Technical Specifications

Before you build a house, you don't just grab a hammer and start nailing boards together. You start with a blueprint — a clear plan of what you're building, who it's for, and how it should function.

Deploying a large language model (LLM) is no different. Before you deploy anything, you need to ask: Why are we building this? What business problem are we solving?

This might seem obvious, but skipping this step is one of the most common causes of failure in AI projects. This is where the journey begins: understanding what your model needs to do, who it's serving, and what success looks like. If you skip this step or do it poorly, you'll end up with a system that's either too slow, too expensive, or just plain unusable.

Imagine you're building a customer service bot. If you don't clarify whether it needs to handle real-time queries, support multiple languages, or integrate with a CRM system, you'll end up choosing the wrong model, overpaying for resources, or delivering a poor user experience.

Let's walk through how to do it right. To do that, we'll start by answering these questions:

- What kind of tasks will the model perform?
- Who is the user?
- What are the performance expectations (e.g., response time, accuracy)?
- What are the cost constraints?

Only once you have clear answers can you move forward with confidence.

Functional requirements: What does the model need to do?

Functional requirements define what your model does — the tasks it must perform and the problems it must solve.

These are the core capabilities your users expect. Think of them as the "features" of your model.

Common use cases and model selection

Here are some typical functional requirements and the models best suited for them:

Here's a handy table to help out:

Use Case	Recommended Model Type	Example
General Q&A	General-purpose LLMs	Qwen, GPT, DeepSeek
Math Problem-Solving	Domain-specific models	Qwen-Math
Medical Diagnosis	Medical LLMs + Knowledge Graphs	Specialized healthcare models
Image Understanding	Vision models	Qwen-VL
Image & Video Generation	Diffusion models	WAN, Stable Diffusion
Voice Assistant	Speech processing models	CosyVoice, Qwen-Audio
Multimodal Tasks	Unified multimodal models	Qwen-VL, BLIP-2

Once you've identified your requirement (or requirements), you'll be faced with another challenge: there's a bunch of models that have similar performance and functionalities (e.g. [Qwen](#), [GPT](#), [Claude](#), [Gemini](#), [DeepSeek](#), etc.). At this stage, you'll need to evaluate them based on your use case - which model performs best under the constraints and provides the best match to your business requirements? You can build your own evaluation model with your own datasets, or use a publicly available dataset to do this. There are also quite a few benchmarks that you can use to test out how well your model performs after you're done with it.



LLMArena's Leaderboard

Source: [LLMArena](#)



Pro Tip:

Don't just pick the *biggest* available model. A smaller, fine-tuned model often outperforms a giant general-purpose one in specific domains — and it's faster and cheaper to run.

How to choose the right model

Once you've identified your requirement (or requirements), you'll be faced with another challenge:

You’ve got dozens of models to choose from. So how do you pick the right one?


Start by asking:

- What kind of task is this? (e.g., summarization, reasoning, code generation)
- Is there a domain-specific model available?
- Do I need high accuracy, or is speed more important?

Then test.

Use real-world data to evaluate models. Build your own evaluation set or use public benchmarks like:

- MMLU – for general language understanding
- BBH – for complex reasoning
- HumanEval – for code generation
- LongBench – for long-context tasks

 **Think Like a Product Manager:**
Your goal isn’t just to pick the most accurate model — it’s to pick the one that best fits your use case, user expectations, and business constraints.

Non-functional requirements: How well should your app do what it's supposed to do?

Once you know what the model needs to do, the next step is to define how well it should do it.

This is where non-functional requirements come in. These are the invisible but critical qualities that determine whether your model feels fast, reliable, and cost-effective to users.

Key Non-functional requirements

Factor	Why It Matters
Performance	Users expect fast responses — especially in chatbots and real-time systems.
Cost	High compute costs can quickly eat into profits or make your product unaffordable.
Stability	If your model crashes or behaves unpredictably, users lose trust.
Security	Sensitive data needs to be protected — especially in regulated industries like finance or healthcare.
Scalability	Can your system handle sudden traffic spikes without breaking?

Performance: Speed is everything


Let’s say you're building a customer service chatbot.

If it takes 5-10 seconds to respond, users will get frustrated. If it responds in the blink of an eye, they’ll think it’s magic.

That’s the power of performance, and the best way to get there is to define a performance baseline, or in expert terms: a Service Level Objective (SLO).

The following table lists some recommended SLOs for different use cases:

Use Case	TTFT (Time to First Token)	TPOT (Time Per Output Token)
Chatbots	< 500ms	< 200ms
Code Completion	< 300ms	< 100ms
Summarization	< 1s	< 300ms
Batch Processing	Varies with use case	Varies with use case

 **Warning:**
Don't guess these numbers. Measure them with real users. What feels fast to you might feel slow to them.

Cost: Deploying smart, not just powerful

You might be tempted to use the most powerful model available — but that can be a costly mistake.


Large models are expensive to run. Every inference call costs money, and those costs add up quickly when you're serving thousands of users.

Instead, optimize.

Use smaller models where possible. Cache common responses. Batch requests. Limit token counts.

And when you're using cloud services, choose your billing model carefully:

Billing Model	Best For	Pros	Cons
Prepaid (Annual/Monthly)	Stable, predictable workloads	Cheapest in the long run	Upfront cost
Pay-as-you-go	Unpredictable usage	No upfront cost	More expensive per use
Spot Instances	Cost-sensitive, fault-tolerant apps	Up to 90% cheaper	Risk of interruption

 **Pro Insight:**
Combine strategies. Use spot instances for background tasks and pay-as-you-go for real-time responses.

Stability: Building a system that never sleeps

Your model might be brilliant, but if it crashes every few hours, nobody will care.

Stability means your system can handle real-world traffic, recover from failures, and keep performing under pressure.

Ask yourself:

- Can your model scale to handle traffic spikes?
- Do you have fallback plans if the model fails?
- Are you monitoring performance and catching issues early?



Think Like an Engineer:

Your goal isn't just to build a model — it's to build a system that can run that model reliably, 24/7.

Performance Optimization: Making Your LLM Responsive

So you've picked the right model. You've defined your performance goals. Now it's time to make that model fast, responsive, and efficient — even under pressure.

Performance isn't just about speed. It's about user experience, cost control, and system scalability. A slow model frustrates users, burns through your budget, and limits how many people you can serve at once.

In this section, we'll explore the most powerful techniques for optimizing LLM performance, from model compression and caching to parallel processing and smart token management. Each strategy is backed by real-world use cases and engineering principles.






Let's get started.


Core tenets of system performance improvement

Think of your LLM as the engine of a high-performance car. It's powerful, but without the right transmission, suspension, and aerodynamics, it won't win any races.

System performance improvement is about tuning the entire stack — not just the model. It's about reducing latency, increasing throughput, and minimizing resource consumption.

Here are the five core principles to follow:

Principle	Description
 Faster (and more efficient) request processing	Reduce inference time through model optimization, quantization, and hardware acceleration.
 Reduce the number of requests	Minimize redundant computations using caching, batching, and deduplication.
 Reduce input and output token count	Trim unnecessary context and constrain outputs to only what's needed.
 Leverage parallel processing	Distribute work across multiple devices or stages to maximize throughput.
 Avoid defaulting to models when simpler solutions exist	Use hardcoding, precomputation, or traditional algorithms where appropriate.

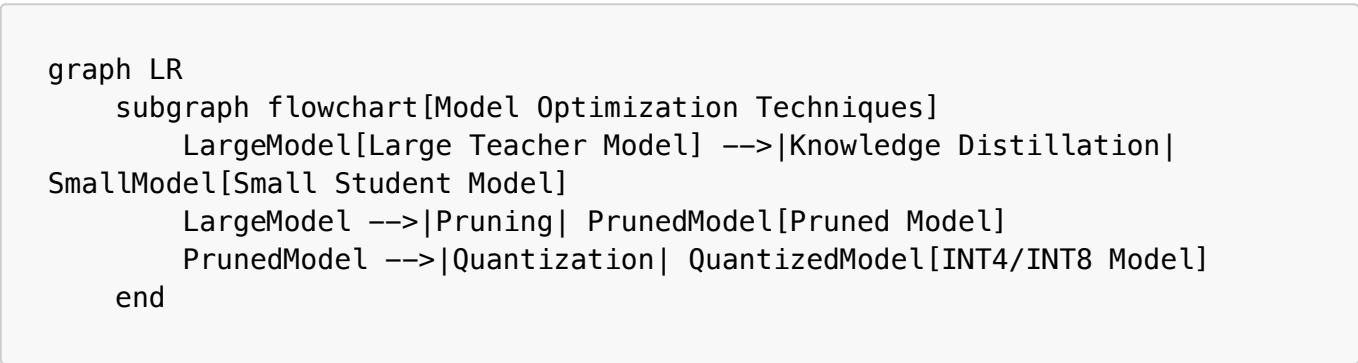
 **Golden Rule:**
The fastest computation is the one you don't have to do.




Faster request processing: make the model think faster (and more efficient)

The single biggest factor affecting inference speed is model size. Smaller models are faster — period.

You don't always need a 600-billion-parameter monster to get great results. In many cases, a well-optimized 7B or 14B model can match or even outperform its larger cousins — especially in domain-specific tasks. As an added bonus, small models take up less resources, which directly translate into saved costs.

So how do small models do it? To oversimplify the answer, small models are typically derived from larger models by various techniques. Here's how:



Technique	How it works
<div>Model pruning</div> <div> Model Pruning</div> <div>Source: Learning both Weights and Connections for Efficient Neural Networks</div>	<p>Removing redundant weights or layers from the model to reduce model complexity.</p> <p>Think of it like trimming a tree to let it grow faster.</p>
<div>Quantization</div> <div> Quantization</div> <div>Source: Qualcomm</div>	<p>Converts FP32 weights to lower precision (e.g., INT8, INT4), reducing memory and compute needs.</p>
<div>Knowledge distillation</div> <div> Knowledge distillation</div> <div>Source: Knowledge Distillation: A Survey</div>	<p>Trains a small "student" model to mimic a large "teacher" model's behavior.</p> <p>Examples:</p> <ul style="list-style-type: none">• DeepSeek-R1-Distill-Qwen-7B• DeepSeek-R1-Distill-Llama-8B

While model compression offers massive gains, there are important trade-offs:

- **Accuracy Loss:** Over-pruning or aggressive quantization can degrade output quality.
- **Domain Limitation:** Distilled models may underperform on out-of-domain inputs.
- **Hardware Compatibility:** Some quantized models require specific GPUs (e.g., NVIDIA A100 for INT4).
- **Retraining Overhead:** Compression often requires retraining or fine-tuning.

Bonus Tip: Optimize Your Prompt

Believe it or not, your prompt can impact performance. A well-crafted prompt can reduce the number of tokens the model needs to process, which speeds up inference.

Try these tricks:

- **Prompt Compression:** Remove redundant instructions.
- **Few-Shot Optimization:** Use concise examples.
- **Prompt Tuning:** Learn a small prefix that guides the model efficiently.

```
graph LR
    Prompt[Original Prompt] --> Compressed[Compressed Prompt]
    style Prompt fill:#f9f,stroke:#333
    style Compressed fill:#bbf,stroke:#333

    subgraph Before
        Prompt -->|200 tokens| Model[LLM Inference]
    end

    subgraph After
        Compressed -->|70 tokens| Model
    end

    Model --> Response[Response]
```

Reduce the number requests (and compute load)

Every inference call costs time and money. So why make the model work harder than necessary?

Smart systems minimize redundant computation by reusing results, grouping requests, and avoiding unnecessary calls.

Caching with context reuse

In applications like chatbots or document analysis, users often provide overlapping context.

Instead of reprocessing the same information repeatedly, cache shared context — such as conversation history or document embeddings.

```
sequenceDiagram
    User->>Server: "Based on our chat, what was my last order?"
    Server->>Cache: Check for cached context
    alt Context Found
        Cache-->>Server: Return cached embeddings
        Server->>LLM: Run inference with cached context
    else Context Not Found
        Server->>LLM: Process full context
        Server->>Cache: Store new context
    end
    end
    Server->>User: Response
```

Impact:

Context caching can cut inference time by up to 50% and reduce GPU usage significantly.

 **Pro Tip:**

Qwen series models support context caching by default. Make sure you're using the right API parameters to take advantage of it.

Batch processing: do more with less

Batching is like doing your laundry in one go instead of one sock at a time. By grouping similar requests together, you reduce overhead and improve hardware utilization.

This is especially useful for:

- Offline summarization
- Data classification
- Report generation
- Data annotation pipelines

```
graph LR
    subgraph Without_Batching [Without Batching]
        R1[Request 1] --> GPU
        R2[Request 2] --> GPU
        R3[Request 3] --> GPU
        GPU --> O1[Output 1]
        GPU --> O2[Output 2]
        GPU --> O3[Output 3]
    end

    subgraph With_Batching [With Batching]
        Batch[Batch: R1+R2+R3] --> GPU
        GPU --> BatchOut[Outputs: O1+O2+O3]
    end
```

Impact:

Batching can improve throughput by 3x or more — and lower GPU costs significantly.

 **Alibaba Cloud Tip:**

Model Studio provides a **Batch API** that you can leverage to run inference during off-peak hours and save on GPU costs.

Reduce input and output tokens: less is more

The more tokens your model has to process, the slower it gets — and the more it costs.

Each token adds compute time — especially during autoregressive decoding (output generation).

So why not reduce the token count?

Input optimization: trim the fat

Don't feed the model everything. Extract only what it needs.

Strategies:

- **Intent extraction:** Use lightweight models (e.g., BERT-based classifiers) to extract core intent before sending to LLM.
- **Summarization preprocessors:** For long documents, summarize first.
- **Rule-based filters:** Remove boilerplate, headers, footers.

Pro Tip:

If you're dealing with long contexts, consider using **hierarchical summarization**. Hierarchical summarization is a powerful technique for reducing the token count of long documents or sequences by breaking them down into smaller, manageable chunks and summarizing them step-by-step. This approach not only speeds up inference but also ensures that the most critical information is retained while minimizing unnecessary details.

```
graph LR
    subgraph flowchart [Hierarchical Summarization]
        direction LR
        LongDoc["Long Document  
(5,000 tokens)"] --> Summarizer1[Summarize  
Sections]
        subgraph Summary_1 [Summary 1]
            Summarizer1 --> MidSummary["Middle Summary  
(800 tokens)"]
        end
        MidSummary --> Summarizer2[Final  
Summary]
        subgraph Summary_2 [Summary 2]
            Summarizer2 --> ShortSummary["Short Summary  
(200 tokens)"]
        end
        ShortSummary --> LLM[LLM Inference]
    end
```

Output optimization: be concise

Generating output is usually the slowest part of the inference pipeline.

So be smart about it:

- Use short prompts like **“Answer in one sentence”** or **“List the top 3 points”**.
- Limit the number of output tokens using the **max_tokens** parameter.
- Avoid open-ended prompts like **Explain everything**.

Real-World Example:

A customer service bot that generates **200 tokens** per response might take 2 seconds. If you reduce it to **50 tokens**, you cut the time — and cost significantly.

Parallel processing: divide & conquer

LLMs are computationally heavy — but that doesn’t mean they have to be slow.

By leveraging parallelism, you can split work across multiple GPUs or even multiple machines, dramatically reducing inference time.

There are three main types of parallelism:

Type	How It Works	Best For
Data Parallelism	Splits input data across devices	Batch inference
Model Parallelism	Splits the model across devices	Large models that don’t fit on a single GPU
Pipeline Parallelism	Breaks inference into stages	Long sequences or multi-stage tasks



Example:

If you're running a 100B model, you might need model parallelism to distribute layers across multiple GPUs. But if you're processing 1000 short prompts, data parallelism will give you the biggest speed boost.

Avoid defaulting to using models: sometimes simple is better

It’s easy to fall into the trap of thinking:

"If a 70B model is good, a 600B model must be better."

But that’s not always true — and it’s rarely cost-effective.

Sometimes, the best performance comes from not using an LLM at all.

Here are some smart alternatives:

Use predefined logic instead of inference

When responses are predictable, repetitive, or based on a finite set of inputs, consider replacing real-time LLM calls with predefined logic. This includes:

- **Hardcoding** for static, high-frequency replies
- **Precomputing** responses for known input patterns

Together, these form a powerful pattern: **"Respond First, Compute Only When Necessary"**.


Hardcoding vs. precomputation: When to use what

Feature	Hardcoding	Precomputation
Definition	Manually writing fixed responses directly in code.	Generating and storing responses in advance for known inputs.

Feature	Hardcoding	Precomputation
Best For	Static, high-frequency responses: <ul style="list-style-type: none">"Invalid input""Request submitted""Hello! How can I help?"	Finite input spaces: <ul style="list-style-type: none">Product FAQsForm validation rulesAPI status codesSupport ticket categories
Example	python if query == "help": return "Available commands: status, help, support" 	json { "refund policy": "Returns within 30 days eligible for refund.", "shipping": "Free shipping on orders over \$50." }
Latency	⚡ Near-zero (~0.1 ms)	🚀 Very low (~1–10 ms with cache/DB lookup)
Cost per Call	💰 Virtually \$0	💰 Close to \$0
Maintenance	Requires code changes to update	Can be updated via config files, DB, or CMS
Scalability	Low — not suitable for 100+ responses	High — scales well with structured data
When to Use	<ul style="list-style-type: none">✅ Response never changes✅ Extremely high frequency✅ Speed is critical	<ul style="list-style-type: none">✅ Input space is limited✅ Responses are predictable✅ Want to avoid real-time inference
Trade-offs	<ul style="list-style-type: none">❌ Hard to manage at scale❌ Requires redeploy to update	<ul style="list-style-type: none">❌ Setup overhead❌ Needs storage/cache layer

Quick decision guide

You Should Use...	If...
Hardcoding	You're responding to common commands like <code>status</code> , <code>help</code> , or <code>greeting</code> — and it's the same answer every time.
Precomputation	You have a catalog, FAQ, or rule set with known inputs and consistent outputs — and you want fast, reliable responses without calling an LLM.

 **Pro Tip:**
Combine both! Use **hardcoding** for core system messages and **precomputation** for domain-specific knowledge (e.g., product details, policies).

Let's try it out real quick:

```
# Define cases
# Hardcoded responses to queries
```

```

HARDCODED_RESPONSES = {
    "status": "Your request has been successfully submitted and is
currently being processed.",
    "help": "Available commands: status, help, support, billing.",
    "greeting": "Hello! How can I assist you today?",
}

# Hardcoded responses for queries
PRECALCULATED_KNOWLEDGE_BASE = {
    "how to reset password": (
        "To reset your password: 1. Go to Settings > Account. "
        "2. Click 'Reset Password'. 3. Check your email for a verification
link."
    ),
    "refund policy": (
        "Our refund policy allows returns within 30 days of purchase. "
        "Visit the Billing section to initiate a refund request."
    ),
    "api documentation": (
        "Full API docs are available at https://api.yourservice.com/docs.
"
        "Includes examples, auth guide, and rate limits."
    ),
}

# Queries for all 3 types of responses
QUERIES = {
    "hardcoded": "status",
    "precomputed": "how to reset password",
    "llm": "Explain quantum entanglement in simple terms"
}

```

```

# Run benchmark for 3 types of responses
benchmark_responses(
    hardcoded_responses=HARDCODED_RESPONSES,
    precomputed_knowledge_base=PRECALCULATED_KNOWLEDGE_BASE,
    queries=QUERIES
)

```

Traditional UI components: Present information visually

Sometimes, the best way to present information isn't with a paragraph — it's with a **structured UI component** like a **table, chart, or progress bar**.

These are:

- **Faster to render** (no long text generation)
- **Easier to scan and understand**
- **More efficient** than generating natural language

Instead of having the LLM describe data in prose, **use its structured output to build a visual summary** — like a Markdown table.


Example:

Replace this LLM text output...

"The user has submitted three orders. Order #1001 was placed on Monday and is pending. Order #1002 was shipped on Tuesday. Order #1003 was delivered on Wednesday. No orders are currently in processing."


With this markdown table

Order ID	Date	Status
#1001	Monday	Pending
#1002	Tuesday	Shipped
#1003	Wednesday	Delivered

 **Why it's better:**

- Users instantly see status and timeline
- No need to parse sentences
- Easier to sort, filter, or extend
- Can be auto-generated from JSON-like LLM output

Even when using an LLM, **prompt it to output structured data** (e.g., JSON, key-value pairs) and **render it as a table** — not a paragraph.

 **Pro Tip:**

If your response includes lists, comparisons, or statuses — **reach for a table before a sentence.**

User Experience Optimization: Beyond Raw Speed

Speed is important — but it's not the only factor that affects user experience. A system can be fast and still feel slow, broken, or frustrating if users aren't kept informed, errors aren't handled gracefully, or feedback loops are missing.

True UX excellence comes from **designing for perception, clarity, and trust** — not just low latency.

In this section, we'll explore practical techniques to make your LLM-powered application feel **responsive, reliable, and user-centric**, even when the backend is still working.

These strategies focus on how the system **communicates with the user** during interaction — creating a smooth, predictable, and reassuring experience.

Basic UX enhancements

Streaming mode: The illusion of responsiveness

Don't make users wait for the full response before showing anything.

Use **streaming mode** to deliver tokens as they're generated, giving the impression that the system is thinking *with* them — not just at them.



Qwen's streaming mode in action

Source: [Qwen](#)

Benefits:

- Reduces perceived latency
- Keeps users engaged
- Feels more conversational and alive

Pro Tip:

Show a subtle "typing" indicator (💬) until the first token arrives.

```
# Query & parameters
query = "Explain why the sky is blue."
system_prompt = """
    You are an educator for 5-year olds. \
    When explaining concepts, make them easy to understand and read. \
    Limit yourself to 5 sentences.
    """

temperature = 0.5
top_p = 0.9

# Streaming response
response, elapsed = get_qwen_stream_response(
    query=query,
    system_prompt=system_prompt,
    temperature=temperature,
    top_p=top_p
)
```

Displaying task progress: Keep users informed

For multi-step tasks (e.g., document analysis, report generation), **break the process into stages** and show progress.

Instead of:

"Processing your request..."

Show:



Analyzing document...



Extracting key points...



Generating summary...

Or use a simple progress bar in Markdown:

[██████████░░░░] 60% Complete – Summarizing content

Why it works:

- Sets clear expectations
- Reduces anxiety during long waits
- Makes the system feel more transparent

Pro Tip:

Pair progress indicators with simple animations if possible. This can help reduce user anxiety by letting them know the system is *responsive*.

```
from IPython.display import clear_output

# Create function to show progress indicator (simulated)
def analyze_document_with_progress(doc, question):
    """
    Analyze a document with simulated progress and final LLM call.
    """
    steps = [
        "📄 Parsing document structure...",
        "🔍 Extracting key entities (products, dates, policies)...",
        "🧠 Determining response type (summary, comparison, FAQ)...",
        "📦 Preparing final prompt for LLM..."
    ]

    # Simulate progress
    for i, step in enumerate(steps):
        print(step)
        time.sleep(0.8)
        clear_output(wait=True)
        progress = "■" * (i + 1) + "░" * (len(steps) - i - 1)
        print(f"Progress: {progress} {int((i+1)/len(steps)*100)}%")
        print(f"Step {i+1}/{len(steps)}: {step}\n")

    print("✅ Starting LLM generation...\n")

    # Final LLM call
    full_prompt = f"""
    Based on the following document, answer the question concisely.
    If the answer isn't present, say "Not found in document."

    Document:
    {doc[:4000]}

    Question:
    {question}
    """
```

```

return get_qwen_stream_response(
    query=full_prompt,
    system_prompt="You are a document analyst. Be accurate and
concise.",
    temperature=0.3,
    top_p=0.9
)

```

```

document = """
    NovaTech Solutions has been a leader in cloud infrastructure since
    2015.
    We serve over 10,000 customers globally with data centers in North
    America, Europe, and Asia.

    Key Offerings:
    - Virtual Machines (NovaCloud VM): Scalable compute instances with GPU
    support.
    - Object Storage (NovaStore): Secure, durable storage with 99.99%
    uptime SLA.
    - AI Training Platform (NovaTrain): End-to-end environment for
    training LLMs.
    - Monitoring Tools (NovaWatch): Real-time dashboards and alerting.

    Recent Updates:
    - Q2 2024: Launched NovaTrain Pro with distributed training support.
    - July 1: All new accounts now include DDoS protection by default.
    - API rate limits are now tiered:
      • Free Tier: 100 requests/minute
      • Pro Tier: 1,000 requests/minute
      • Enterprise: Unlimited with priority routing

    Support: Available via email (support@novatech.com) and live chat (9
    AM – 6 PM EST).
    """

```

```

# Analyze & query the document, showing the progress
question = "What are the key offerings of NovaTech?"
response, t = analyze_document_with_progress(document, question)

```

Robust error handling: Gracefully handle the unexpected

Errors happen — but how you present them defines user trust.

Avoid raw error messages like:

```
Error: Model inference failed (status 500)
```

Instead, show user-friendly, actionable feedback:

✗ We couldn't process your request right now.
Please check your input and try again.
If the issue persists, contact support.

And log the technical details server-side.

Key Principles:

- Never expose internal system errors
- Offer a clear next step
- Maintain tone and brand voice

Pro Tip:

Automatically retry transient errors (e.g., timeout) before showing any error to the user.

Other UX enhancements

Implementing feedback mechanisms: Let users feel heard

You can implement feedback mechanisms to let users feel heard by letting them rate responses directly. Users are able to express their views and provide suggestions (this is especially important if your LLM caters to a specific niche). A typical feedback mechanism might look something like this:

Was this response helpful?
👍 Yes 👎 No

This data can be used to:

- Identify weak spots in your pipeline
- Retrain or fine-tune models
- Route low-confidence queries to human review

Smart Follow-ups: Suggest what the user might want to do next

After delivering a response, don't leave the user hanging. A great LLM application anticipates the next step and guides the conversation forward.

Why it matters:

- Proactive suggestions reduce friction and increase engagement.
- It mimics natural human conversation — where one question leads to another.

Reliability & Resilience: Building Systems That Don't Break Easily

You've built a smart app — now make sure it can survive real-world traffic, failures, and spikes in usage. In production, reliability is just as important as accuracy.

Let's break reliability & resilience for your app down into two pillars: **Infrastructure Management** and **Request Handling**.

Infrastructure: The foundation of stability

Baseline management

Establishing a performance baseline is a critical first step in any LLM application lifecycle. Instead of jumping straight into complex models, teams should begin with simple, interpretable approaches—like rule-based systems or lightweight algorithms—to set a minimum performance threshold. This baseline acts as a reference point for evaluating future models, ensuring that upgrades actually improve results. By regularly comparing new models against this benchmark across time and real-world scenarios, and integrating automated checks into deployment pipelines, organizations can maintain model reliability, adapt to changing conditions, and safely roll out improvements through strategies like canary releases.

Other baseline management options:

- Model versioning: Pin your model versions and avoid auto-upgrades.
- Environment parity: Ensure dev, staging, and prod environments are as similar as possible.
- Configuration as code: Store configs (prompts, parameters, API keys) in version-controlled, encrypted secrets.

Pro Tip:

Use feature flags to toggle between model versions or prompt variants without redeploying.

Auto scaling

LLM workloads are bursty — users don't ask questions at a steady rate. Scale intelligently:

- Horizontal scaling: Spin up more inference instances during peak hours.
- Serverless options: Use platforms like Alibaba Cloud's Model Studio to scale to zero when idle.
- Queueing: For long-running tasks, use message queues (e.g., RabbitMQ, Kafka) to decouple request and processing.

Example:

During onboarding week, your HR bot gets 10x traffic. Auto-scaling ensures responses stay fast — without overpaying during quiet periods.

Disaster recovery

Assume failure will happen. Plan for it.

- Fallback models: If your primary LLM (e.g., Qwen-72B) is down, route to a smaller, faster model (e.g., Qwen-1.8B).
- Circuit breakers: Temporarily halt LLM calls if error rate exceeds threshold (e.g., 50% timeout).
- Data backups: Regularly back up your RAG index, logs, and user feedback.

Recovery Goal:

"If the LLM goes down, the system should degrade gracefully — not fail catastrophically."

Handling: Managing the flow of requests

Rate limiting

Prevent abuse and ensure fair usage.

- **Per-user limits:** e.g., 100 queries/hour for free users, 1,000 for premium.
 - **Token-based throttling:** Limit by input + output tokens, not just requests.
 - **Dynamic limits:** Adjust based on system load or user role.
- 🔴 Why? Without rate limiting, one user can saturate your LLM quota and slow down everyone else.

Monitoring & error handling

You can't improve what you don't measure. Here are a few metrics that you can use to monitor the performance of your LLM over extended period of time:

Metric	What it tells you
Latency	Is inference slowing down?
Error Rate	Are we hitting timeouts or 5xx errors?
Token Usage	Are prompts getting too long?
Cache Hit Ratio	Are we reusing responses effectively?

Error handling best practices:

- **Return graceful fallbacks:** "I can't answer that right now, but here's a link to the help page."
 - **Log full context:** input, model version, response time, and error message.
 - **Alert on anomalies:** e.g., sudden spike in hallucination rate or latency.
- Pro Tip:**
Treat your LLM like a microservice — because it is. Apply the same SRE (Site Reliability Engineering) principles: SLAs, SLOs, and post-mortems.

Cost Considerations: Building Financially Prudent Applications

Improving performance can reduce costs

Many of the performance improvements we went over in the previous sections not only reduce latency but also significantly lower operational costs. Let's do a quick recap:

- **Use smaller models where possible:** Smaller models offer faster inference and are inherently less expensive to run, making them ideal for tasks that don't require the full capacity of large models.

- **Context caching:** Cache results for frequently repeated queries to avoid redundant LLM calls. For example, on Alibaba Cloud, the **cache_token** cost for Qwen models is only 40% of the **input_token** cost, making caching a highly effective cost-saving measure.
- **Batch inference:** Combine or deduplicate requests to minimize unnecessary model invocations. Batch processing is typically suitable for non-real-time tasks and can be scheduled during off-peak hours to leverage idle compute resources, further reducing costs. For instance, using Alibaba Cloud's Bailian platform, batch inference is priced at just 50% of real-time inference.
- **Reduce token usage:** Minimizing input and output token counts directly reduces computational load, lowering both hardware and energy costs.
- **Avoid over-reliance on large models:** Offload simple or deterministic tasks (e.g., rule-based logic, data formatting) to traditional code or precomputed systems instead of using expensive LLM inference.

These strategies are broadly applicable. In real-world deployments, two primary deployment models exist: on-premises infrastructure and cloud-based deployment. On-premises solutions involve high upfront costs, long procurement cycles, and complex maintenance. In contrast, cloud deployment is better suited for startups and cost-sensitive applications. Cloud platforms offload infrastructure management to the provider and enable rapid scaling, ensuring efficient resource utilization.

Cost optimization on the cloud

LLMs are typically associated with massive volumes of data storage, high-performance computing, and complex inference workloads, leading to potentially high operational costs. This section explores how to make cost-effective architectural choices when deploying LLM applications on the cloud.

Selecting the right GPU-accelerated instance

When deploying LLMs on Alibaba Cloud, choosing the appropriate ECS instance type—such as single-GPU, multi-GPU, or distributed multi-node setups—requires balancing performance needs with budget constraints. Consider not only the model's memory footprint but also runtime resource demands like KV Cache:

- **Model parameter count:** The number of parameters directly determines GPU memory requirements. As introduced in earlier chapters, a 1.5B-parameter model requires approximately 5.59 GB of memory (FP32 precision), while the non-minimized **DeepSeek-R1** (671B parameters) needs at least $\frac{671 \times 10^9 \times 2^{30}}{1024} \approx 625$ GB (FP8 precision).
- **KV Cache overhead:** In generative tasks (e.g., text generation), the KV Cache stores attention key-value pairs. Its size scales with sequence length and batch size. For example, processing a 2048-token context can consume significant GPU memory.
- **Precision settings:** Different precision levels (FP32, FP16, INT8, INT4) impact memory usage. Quantization techniques can dramatically reduce memory footprint.

Let's use **DeepSeek-R1 (671B)** as an example. At FP8 precision, the model itself requires ~625 GB of memory. With KV Cache optimized via MLA (Multi-head Latent Attention), assuming each token uses ~70 KB of cache per GPU, an 8-GPU system handling a 64K-token request would need ~35 GB for KV Cache.

Total memory required: ~660 GB. This is a good match for the **ecs.ebmgn8v.48xlarge** instance (8 × 96 GB = 768 GB total).

The maximum number of concurrent requests is constrained by GPU memory:

$$\frac{\text{req} \times l \times (70 \times 1024)}{2^{30}} + \frac{625}{8} < 96 \text{ GB}$$

Where req is the number of concurrent requests and l is the token length.

Assuming $l = 64K$, we find $\text{req} \approx 4$. Due to tight memory constraints, the system can serve only a few users simultaneously. To support higher concurrency, consider limiting input length or upgrading to a higher-capacity GPU instance.

Choosing the right billing model

Once you’ve selected your instance, choose a billing model based on your business needs:

	Subscription	Pay-As-You-Go	Spot Instances
Description	Prepay for reserved resources and enjoy significant discounts. Ideal for long-term, stable workloads.	Use resources on demand without upfront payment. Flexible and fast to deploy. For medium-to-long-term usage, combine with Savings Plans to reduce costs.	Pay after use. Pricing fluctuates based on supply and demand, offering up to 90% savings compared to pay-as-you-go.
Recommended Use Cases	Stable, long-term deployments (e.g., months or years)	Short-term or variable-duration workloads	Non-critical workloads where brief interruptions are acceptable, and cost is a top priority. Best paired with retry mechanisms or fault-tolerant designs. For online inference with spot instances: Best practices for PAI-EAS Spot

We also recommend using Budget Management to set cost alerts and regularly reviewing Cost Analysis to identify high-cost components (e.g., GPU/ECS instances). Test different configurations for cost-performance trade-offs, prioritize core functions (e.g., retrieval) under high load, and disable non-essential services (e.g., complex generation tasks) to optimize spending.

What's next?

Quiz yourself!

► **1. What is the primary benefit of using a smaller, distilled model over a larger general-purpose one?**

- A) Higher accuracy on all tasks
- B) Faster inference, lower cost, and better domain performance
- C) Larger context window
- D) No need for fine-tuning

View answer →

✓ **Correct answer:** B) Faster inference, lower cost, and better domain performance

📝 **Explanation:**

Smaller models, especially those distilled for specific domains (e.g., Qwen-Math), often outperform larger models on targeted tasks while consuming fewer resources. This leads to faster responses and reduced operational costs.

► **2. Which technique reduces inference cost by reusing previously computed context?**

- A) Model pruning
- B) Context caching
- C) Prompt compression
- D) Data parallelism

View answer →

✓ **Correct answer:** B) Context caching

📝 **Explanation:**

Context caching stores embeddings or key-value states from prior interactions, allowing the model to skip reprocessing shared context. On platforms like Model Studio, this can reduce input token costs by up to 60%.

► **3. What is the "Golden Rule" of LLM performance optimization?**

- A) Always use the largest model available
- B) Maximize token output for completeness
- C) The fastest computation is the one you don't have to do
- D) Stream every response regardless of use case

View answer →

✅ **Correct answer:** C) The fastest computation is the one you don't have to do



Explanation:

This principle underpins caching, hardcoding, precomputation, and input trimming — all strategies that avoid unnecessary LLM calls to improve speed and reduce cost.

Takeaways

- **What makes a production-ready LLM app**
 - **It starts with requirements** — define functional (what it does) and non-functional (how well it does it) needs upfront.
 - **Speed matters** — users expect sub-second responses. Define SLOs (e.g., TTFT < 500ms for chatbots).
 - **Cost scales fast** — every token and inference call adds up. Optimize early.
 - **Stability is non-negotiable** — your system must handle failures gracefully (fallbacks, circuit breakers, monitoring).
- **Performance optimization is a stack-wide effort**
 - **Faster processing:** Use smaller, quantized, or distilled models.
 - **Fewer requests:** Cache context, deduplicate queries, precompute responses.
 - **Fewer tokens:** Trim input, limit output length, use summarization preprocessors.
 - **Parallelism:** Use data/model/pipeline parallelism to scale throughput.
 - **Avoid LLMs when possible:** Hardcode or precompute deterministic responses.
- **User experience goes beyond latency**
 - **Streaming** creates the illusion of speed — show tokens as they arrive.
 - **Progress indicators** keep users informed during multi-step tasks.
 - **Graceful errors** maintain trust — never show raw stack traces.
 - **Feedback buttons** (👍/👎) provide data for improvement.
 - **Smart follow-ups** guide the conversation forward naturally.
- **Reliability requires engineering discipline**
 - **Auto-scaling** handles traffic spikes without over-provisioning.
 - **Rate limiting** prevents abuse and ensures fair usage.
 - **Monitoring** tracks latency, errors, token usage, and cache hit ratios.
 - **Disaster recovery** includes fallback models, circuit breakers, and data backups.
 - **Treat your LLM like a microservice** — apply SRE practices (SLAs, SLOs, post-mortems).
- **Cost optimization is strategic**
 - **Smaller models are often better** — especially when distilled or fine-tuned.
 - **Caching cuts costs** — reused context is cheaper than reprocessing.
 - **Batching saves money** — off-peak batch jobs can cost 50% less.

- **Choose billing wisely** — use subscription for stable loads, spot instances for fault-tolerant tasks.
- **Measure everything** — use cost analysis tools to identify spending hotspots.

- **The best system is often the one that doesn't call the LLM**
 - **Hardcoding** is near-zero latency and cost-free — ideal for common replies.
 - **Precomputation** trades setup effort for runtime efficiency — perfect for FAQs.
 - **Structured UIs** (tables, charts) are faster and clearer than prose.
 - **Rule-based filters** can route or resolve queries before they reach the model.