

LLMs in Production:

What Breaks, What Works, and Why

A practical mental model for developers, POs, QA and everyone building with AI

*Building on recent sessions about coding conventions, agents & MCP —
this talk explains the WHY behind those practices.*

Agenda

- | | | |
|----|---|-----------|
| 01 | Hook: Why LLMs frustrate us | 0–3 min |
| 02 | The core mental shift: Deterministic vs Probabilistic | 3–8 min |
| 03 | LLMs as a workflow accelerator | 8–15 min |
| 04 | Embedding LLMs in your product | 15–23 min |
| 05 | Concrete demo: Digital signage example | 23–28 min |
| 06 | Takeaways & strategic implications | 28–30 min |

Why Do LLMs Frustrate Us?

You've probably experienced at least one of these...



It wrote perfect code... then failed on the second attempt with the same prompt



It hallucinated an API, a library, or a function that simply doesn't exist



It 'forgot' a constraint you gave it 10 messages earlier in the same chat



It gave brilliant output today and confusing output tomorrow — same question



It confidently told you something wrong, with zero indication of uncertainty

The Core Mental Shift

Deterministic code vs. Probabilistic systems

Deterministic Code

Same input

→ always same output

Explicit control flow

traceable step by step

Unit-testable

pass / fail is reliable

Precise

errors are deterministic bugs

Best for

logic, routing, validation, data

LLM (Probabilistic)

Same input

→ distribution of outputs

Implicit statistical reasoning

not explicit logic

Must validate externally

output is never guaranteed

Approximate

confident \neq correct

Best for

language, generation, reasoning



Most LLM frustration = expecting deterministic behaviour from a probabilistic system

LLM as a Workflow Accelerator

Augmenting how you work — not replacing what you build

This is WHY the coding conventions from recent sessions work — here's the underlying principle:



Think Before You Prompt

Clear plan + AI = fast.

No plan + AI = fast in the wrong direction.
plan.md → tasks.md → then prompt.



Structure Beats Chat

spec.md, prd.md, interfaces.ts first.

Give the LLM a framework — it replicates patterns it can clearly see.



Iterate Small

'Create the interface' → 'Implement'
→ 'Write the tests' beats one massive prompt every time.



Log Everything

Full system prompt. Full user message.
Full raw response. If you don't log,
you're guessing, not debugging.



Validate Output

Never trust blindly. Parse, schema-check,
test. LLM output is a first draft —
not a finished result.



Manage Context

Context windows are finite.
The LLM only knows what's in the window.
Summarise earlier turns explicitly.

Embedding LLMs in Your Product

A very different problem from using one yourself

Relevant for Digital Signage

- Dynamic promotional copy on displays
- Natural language playlist / scheduling
- Auto-tagging and categorising media
- Chatbot or assistant inside the platform
- Summarising content feeds automatically

Key Engineering Concerns

Latency

0.5–5s per call. Do screens wait? Async / stream.

Cost

Tokens = money. Bad prompts = cost explosion.

Routing

Small model for classification. Large for reasoning.

Context limits

Finite window. Design for overflow from day one.

Guardrails

Validate JSON. Sanitize. Moderate before display.

Fallbacks

API down → do screens go blank? Design for it.

Concrete Demo: Digital Signage Prompt

Retail screen — generate promotional copy dynamically

✗ NAIVE PROMPT

User prompt:

```
Write a promotional message for a summer sale.
```

Output:

```
"Summer is here! Enjoy amazing deals  
on all our great products. Shop now!"
```

- ✗ Vague & generic
- ✗ Brand-inconsistent
- ✗ No structure — can't parse
- ✗ Why people think LLMs are unreliable

✓ STRUCTURED PROMPT

System prompt (developer-set, user never sees):

```
You are a copy generator for retail digital signage.  
Output: valid JSON only. Max 20 words.  
Tone: energetic but premium. No emojis.  
Audience: Scandinavian fashion retail.
```

User prompt:

```
30% summer sale — linen shirts.
```

Output:

```
{ "headline": "30% Off Linen Essentials",  
  "body": "Lightweight summer styles crafted  
          for comfort and elegance." }
```



This is not prompt engineering. This is constraint engineering — and now it's a product component.

Key Takeaways

What to carry out of this room



LLMs are probabilistic engines — not magic, not databases, not a reliable function



Structure, naming, interfaces and plans work because they reduce ambiguity for a probabilistic system



If you don't log full prompts + full responses, you are guessing — not debugging



Using an LLM yourself vs. shipping one in a product are fundamentally different engineering problems



Deterministic layers around probabilistic cores = robust, production-grade systems



Know when to say no — real code beats an LLM for deterministic, latency-critical, or high-stakes tasks



Happy to stay and go deeper — bring your questions, examples, and frustrations!