Architectural Patterns for Integrating LLMs into User-Facing Applications

Lessons from a language-learning platform

MIRCEA LUNGU, IT University of Copenhagen, Denmark

CESARE PAUTASSO, University of Lugano, Switzerland

Large Language Models are increasingly being integrated as components into existing user-facing applications, alongside the more visible wave of LLM-native products such as chatbots and agents. The engineering concerns of the two cases differ: when an LLM is a component behind an existing feature rather than the product itself, designers must reconcile its per-token cost, multi-second latency, non-determinism, imprecision, rapidly shifting provider landscape, and general-purpose capability with the expectations of users who already rely on the system. We present a catalogue of recurring architectural patterns that address these concerns, grounded in over a year of LLM integration work on Zeeguu, an open-source language-learning platform with several hundred monthly active users. The patterns are grouped into three themes (using the LLM efficiently, trusting its output, and managing change over time) and described in the standard context, forces, solution, and consequences format. They are presented as a starting point for community refinement and extension, not as a closed taxonomy.

Additional Key Words and Phrases: large language models, software architecture, design patterns, LLM integration, cost, latency, quality assurance

1 Introduction

While there is growing literature on building LLM-native products such as chatbots, agents, and retrieval-augmented generation (RAG) pipelines, and on using LLMs for code generation, there is surprisingly little guidance on the **software engineering challenges of integrating LLMs as components into existing interactive applications** where real users expect fast, reliable, and trustworthy responses.

We expect this integration to become the common case: over time, more and more existing user-facing applications will adopt LLMs not as standalone chatbots but as components working behind the scenes to improve the user experience. This is why we present a set of patterns that highlight the challenges and opportunities such integrations raise.

Indeed, LLMs have a unique combination of properties:

- **expensive**: billed per token, with a large fixed prompt re-paid on every call;
- **slow**: responses take seconds, not the milliseconds an interactive interface expects;
- **non-deterministic**: the same input can return a different answer;
- **rapidly evolving**: models are released and retired on the vendor's schedule, every few months;
- **imprecise**: they make mistakes, from a malformed response to a confident but wrong answer; and
- **general-purpose**: they can attempt almost any task expressed in text.

Integrating a component with these properties into a live, user-facing system is what creates the architectural forces these patterns resolve: the tension between what the model is (slow, costly, non-deterministic) and what users and the surrounding code expect (fast, affordable, well-formed).

Authors' Contact Information: Mircea Lungu, IT University of Copenhagen, Copenhagen, Denmark, mircea.lungu@gmail.com; Cesare Pautasso, University of Lugano, Lugano, Switzerland.

Over the past year, we have been integrating LLMs into Zeeguu[1], an open-source platform for personalized language learning that helps users learn foreign languages by enabling them to read authentic online content (real articles, not textbook exercises). Through this work, we have identified a set of architectural patterns for LLM integration, several of them corroborated by documented use in other systems. Because these forces arise from the LLM's own properties and the generic demands of a live application, not from anything specific to language learning, we expect them, and the patterns that resolve them, to recur wherever LLMs are integrated into user-facing systems.

The remainder of the paper is organised as follows. Section 2 introduces Zeeguu, the platform that grounds every example. The catalogue then follows in three themes, using the LLM efficiently (Section 3), trusting its output (Section 4), and managing change over time (Section 5), each pattern given in the same format: context, example, problem, forces, solution, consequences, known uses, notes. Section 6 asks what makes these patterns specific to LLMs, and the paper closes with related work, limitations, and conclusions.

2 Case Study: Zeeguu

Zeeguu is an open-source language learning platform built around the idea that learners benefit most from engaging with comprehensible [11] and authentic [7] content in their target language. Rather than relying on artificial textbook texts and exercises, Zeeguu helps users find real articles (news, blog posts, and other web content) tailored to both 1) their *level* and 2) their *interests*. Then, based on the words they don't understand, it generates personalized vocabulary exercises and audio lessons.

The platform recommends articles in the learner's target language based on their proficiency and reading preferences, making it **easy to find material that is both engaging and appropriately challenging**. If a text is personally compelling but too difficult, Zeeguu simplifies it to the learner's level using LLMs.

When users encounter unfamiliar words or phrases while reading, they can get contextual translations on the fly from several translation providers, so the reading experience remains fluid and uninterrupted. One alternative, available on demand, comes from a state-of-the-art LLM offering a more contextually nuanced option (the "Ask an AI" escalation described below).

Every translation a user requests is logged by the system, which over time builds a **detailed model of the learner's vocabulary knowledge**, tracking which words they know, which ones they struggle with, and how well they've retained previously encountered vocabulary.

Based on this evolving learner model, Zeeguu **generates interactive vocabulary exercises and audio lessons** that focus on the words that matter most for each individual learner, rather than following a generic curriculum. The exercises use the context in which the word was originally encountered, based on the assumption that if the original text was compelling to the learner, examples drawn from it will be too.

In essence, Zeeguu unifies reading, translation, learner modeling, and practice into a coherent pipeline, with the learner's own reading interests as the primary driver.

Zeeguu also supports teacher accounts: a teacher can assign texts to a class and follow each student's reading and exercises. Because a teacher's judgement is more authoritative than an individual learner's, their corrections act as a higher-trust signal in the learner model.

[1] https://zeeguu.org

2.1 The Pieces

A handful of Zeeguu-specific concepts recur across the patterns. They are collected here once, so a pattern can point back to a single definition rather than re-explaining them.

2.1.1 CEFR Levels. The Common European Framework of Reference grades language proficiency on an ordered six-level scale, from A1 (beginner) to C2 (mastery). Zeeguu uses it in two directions: to estimate how hard an article is, and to simplify an article down to the level of a given learner.

2.1.2 Article Simplification. When an article is compelling but too hard, Zeeguu rewrites it with an LLM to easier CEFR levels, producing one simplified version per level below the original. It runs both on demand, when a reader opens an article that has not been simplified yet, and ahead of time, over the crawled feed for some articles deemed likely to appeal to readers.

2.1.3 Crawling. Zeeguu builds its article recommendations by crawling news sites and blogs multiple times per day; this steady feed of freshly crawled articles is where most ahead-of-time LLM work happens (CEFR assessment, simplification), off any reader's critical path. On demand, readers push their own content in: a browser extension sends any article to Zeeguu for study, and on mobile the system share sheet sends any web page to be made interactive and studied within the application.

2.1.4 Multi-Word Expressions. A multi-word expression (MWE) is a group of words whose meaning is not the sum of its parts, for example *break the ice*. Zeeguu detects them so a learner can translate the phrase as a unit rather than word by word. A cheap dependency-parse gate (Stanza [17], an open-source NLP parser) fires first, and an LLM confirms only the flagged sentences.

2.1.5 Meanings and The Learner Model. A *meaning* is a word paired with one particular translation of it. A meaning is linked to the original context in which it was seen, but can also be associated with other example sentences in which the word carries that same translation. Every translation a learner requests is logged as a meaning, building a model of which meanings they know and which they struggle with, since one word can have several meanings. Meanings are classified (by frequency, CEFR level, and phrase type: single word, collocation, idiom, or expression) and drive which exercises and lessons the learner sees. Vocabulary training trains *meanings*, not words.

2.1.6 Translation. While reading, a learner gets an instant contextual translation that is generated with the help of multiple parallel translation APIs. If they all agree, the translation is inserted above the word. If they disagree, a popup surfaces the disagreement along with the option to escalate to an LLM through the "Ask an AI" action. This is one extra, more expensive step, offered for the cases where it is unclear which of the translations is correct.

2.1.7 Audio Lessons. Zeeguu generates personalized audio lessons: an LLM writes a short script built around the words a learner is studying, and text-to-speech synthesizes the audio. Both steps are slow and costly, so lessons are pre-computed for recently active learners. Once a learner chooses a topic (European history) or a situation (talking to an elderly neighbour on the stairway), a new lesson is generated each day, conditional on the learner having listened to the previous day's lesson.

2.1.8 Vocabulary Exercises. Exercises are built from the words a learner has looked up. They use example sentences drawn from the learner's own reading where possible, and LLM-generated, then LLM-validated, sentences otherwise.

Original context is used only where possible, because not all contexts in which a learner encountered a word suit an exercise: some are too long, others do not make sense outside their original context.

2.2 Reach

Zeeguu currently serves over 300 monthly active users across 11 languages (Danish, Dutch, English, French, German, Greek, Italian, Portuguese, Romanian, Spanish, and Swedish), with peaks exceeding 400 during the academic year[2]. It is publicly available to try: the web app is at zeeguu.org[3] with the invite code `zeeguu-beta`, and the mobile apps can be downloaded from the iOS and Android app stores.

3 Using the LLM Efficiently

An LLM call is slow and metered, so a recurring question in any integration is how *not* to make the call, or how to make each one count. The patterns in this section keep the model's cost and latency off the user's path: paying a large fixed prompt once across a batch rather than once per item, reaching for the LLM only when a cheaper tool falls short, letting a classical stage gate it so it runs only where its judgment is needed, and computing likely-needed results ahead of time so the model never runs while a user waits. The unifying force is that the LLM is the expensive, slow component and the cheapest call is the one that is never made.

3.1 Prompt Amortization

3.1.1 Context. Many LLM calls share the same shape: a large, fixed instructional prompt (rules, taxonomies, output format, examples) wrapped around a tiny variable input (see figure). The work is offline and batchable: nobody is blocked on any single result.

3.1.2 Example. Several Zeeguu jobs have exactly this shape. Rather than pay the preamble once per item, they batch (i.e., pack) related items into a single call, in one of two directions: **fan-in** (many inputs, one call) or **fan-out** (one input, many outputs):

- **Meaning (Section 2.1.5) classification** (*fan-in*) sends ~15 word-meanings per call, sharing one frequency/CEFR-type taxonomy prompt across the whole batch.[4]
- **Example-sentence validation** (*fan-in*) checks ~20 generated examples per call.[5]
- **Article simplification (Section 2.1.2)** (*fan-out*) produces every CEFR level simpler than the original in one call, one section per level, turning four or five requests into one, about 75% fewer calls for a typical article.[6]

3.1.3 Problem. Sent one item at a time, that fixed preamble is re-paid on every call and dominates token cost and latency. How can it be paid once instead of once per item, without blowing past the model's quality or context limits?

[2]Monthly active users are defined as users with any learning activity (exercises, reading, browsing, audio lessons, or translations) in a given month. Live statistics are available at: https://api.zeeguu.org/stats/monthly_active_users
[3]https://zeeguu.org
[4]The batched meaning-classifier prompt, `create_batch_meaning_frequency_and_type_prompt` in the `zeeguu/api` repository.
[5]The batch example-sentence validator, `validate_examples_batch` in the `zeeguu/api` repository.
[6]The multi-level simplification prompt, `get_adaptive_simplification_prompt` in the `zeeguu/api` repository.

```
COMBINED_VALIDATION_PROMPT = """You are validating and classifying a language learning translation.

A learner highlighted "{word}" in this {source_lang} sentence:
"{context}"

The translation is in {target_lang}: "{translation}"

CRITICAL: The translation "{translation}" is in {target_lang}, NOT English!
If {target_lang} is Dutch, German, etc., interpret the translation IN THAT LANGUAGE.
Example: "offer" in Dutch means "victim/sacrifice" - this is VALID for "víctima" (Spanish).

STEP 1 - VALIDATION:
1. Is "{word}" a meaningful learning unit? (Not an arbitrary fragment like "bruger den" = "are using it")
2. Is "{translation}" a correct translation for the WORD itself (not the surrounding phrase)?
3. If the word is part of an idiom, decide: should the learner study the single word or the full idiom?

IMPORTANT: Only mark as FIX if the translation is WRONG. Do NOT "improve" correct translations.
- Keep translations SHORT and SIMPLE - learners must type them in exercises
- NO parenthetical explanations like "by (doing something)" - just "by"
- NO alternatives with "/" like "as/in the capacity of" - pick ONE: "as"
- NO articles unless essential: "eyes" not "the eyes"

STEP 2 - CLASSIFICATION (for the valid/corrected word):
Frequency:
- unique: only meaning of the word
- common: primary or frequently used meaning
- uncommon: infrequently used meaning
- rare: specialized, archaic, or context-specific

CEFR Level (what level learner should know this word):
- A1: basic everyday words (hello, cat, eat)
- A2: common everyday vocabulary (weather, shopping)
- B1: intermediate topics (opinions, work, travel)
- B2: abstract concepts, nuanced vocabulary
- C1: advanced, sophisticated vocabulary
- C2: rare, literary, highly specialized

Phrase type:
- single_word: individual word
- collocation: natural word combination ("strong coffee", "take place")
- idiom: non-literal meaning ("break the ice", "piece of cake")
- expression: common phrase/greeting ("how are you")
- arbitrary_multi_word: random fragment, NOT worth studying ("doctor for", "the cat on", "bruger den")

Reply in this EXACT format (one line):
VALID|frequency|cefr|phrase_type|explanation|literal_meaning
or
FIX|corrected_word|corrected_translation|frequency|cefr|phrase_type|reason|explanation|literal_meaning

Fields:
- cefr: A1, A2, B1, B2, C1, or C2
- explanation: OPTIONAL usage notes, register (formal/informal). Leave empty if not needed.
- literal_meaning: ONLY for idioms - SHORT word-by-word translation (e.g., "kick into touch").
  Leave EMPTY for single words, collocations, and non-idioms. NOT for explanations!

Examples:
- Simple word: VALID|common|A1|single_word||
- Word with usage note: VALID|common|B1|single_word|formal register|
- Idiom: VALID|common|B2|idiom||kick into touch
- Wrong translation: FIX|øjnene|the eyes|common|A2|single_word|literal meaning is 'the eyes'||
- Idiom fix: FIX|se virkeligheden i øjnene|face reality|common|B2|idiom|idiom meaning 'face reality'||see reality in the eyes
- Arbitrary fragment: FIX|bruger|use|common|A2|single_word|'bruger den' is arbitrary fragment||
"""
```

Fig. 1. The `COMBINED_VALIDATION_PROMPT` template: ~250 lines of validation rules, frequency/CEFR/phrase-type taxonomies, output format, and examples, wrapped around just three variables (`{word}`, `{translation}`, `{context}`). Sent one pair at a time, the entire preamble is re-paid on every call. This fixed overhead is the cost the pattern amortizes.

3.1.4 Forces.

- **Preamble overhead.** The preamble is large by necessity: detailed rules, taxonomies, and examples are what make the output accurate and consistent, so quality pulls it up, and it cannot be trimmed without losing quality. Sent one item at a time, that fixed preamble is re-paid on every call, in tokens, cost, and latency. The only lever left is to amortize it. *(pushes toward bigger batches)*
- **Quality ceiling.** Accuracy and consistency degrade as more items share one call; past ~15–20 small items some models start dropping or muddling entries. *(pushes toward smaller batches)*
- **Context ceiling.** Input *and* output must fit the context window; for fan-out the output side binds first, since each result is full-length.

3.1.5 Solution. At its core, this is `map` for LLM calls: apply one shared, expensive prompt across a batch so its setup is paid once, not once per item. It takes two forms:

- ***Fan-in* batching** packs many independent inputs into one prompt, spreading a large instructional preamble across the whole batch.
- ***Fan-out* batching** produces many outputs from a single input, emitting one section per output and collapsing several requests into one.

Both combine naturally with *Anticipatory Precomputation* (computing likely-needed results ahead of time): because results are computed offline, there is the luxury of batching.

3.1.6 Consequences.

- **Cost and latency amortize with batch size.** The fixed preamble is paid once per call instead of once per item, so per-item token cost *and* wall-clock latency fall roughly inversely with the batch size.
- **Batch size is capped by the tightest ceiling, and which ceiling binds flips by direction.** For fan-in (many small items) the *quality* ceiling binds first (~15–20 items before the model drops or muddles entries), far below what the token window allows; for fan-out (full-length outputs) the *token* ceiling binds first, on the output side, since each result is full-length. So the workable batch size is workload-specific and must be tuned, not maximized.
- **Applies only to deferrable work.** Fan-in must wait to accumulate items, so the pattern fits offline / pre-computed paths (composes with *Anticipatory Precomputation*) and is unavailable when a user is blocked on a single result.

3.1.7 Known Uses.

- *The fan-in direction is prior art.* Packing many independent inputs under one shared prompt is the published **batch prompting [3]** technique (Cheng, Kasai & Yu, EMNLP 2023), which cuts token and time cost roughly inverse-linearly with batch size. This pattern's contribution is to add the *fan-out* direction and to frame both as a single move: amortizing one fixed preamble.
- The *fan-out* direction maps directly to structured-output calls that emit several keyed results at once, and to the OpenAI/Anthropic multi-output `n` parameter (which requests several completions per call).

3.1.8 Notes.

- **Both directions can be conceptualized as the functional programming concept `map`, over a different axis.** Fan-in maps the prompt over inputs, e.g. `map(validate, examples)`. Fan-out maps over outputs for a fixed input, e.g. `map(level -> simplify(article, level), levels)`. Either way the shared prompt is the function whose fixed setup the batch pays for once.
- **Two adjacent provider mechanisms are not this pattern.** *Provider batch APIs* (OpenAI Batch[7], Anthropic Message Batches[8]) give ~50% off large asynchronous jobs, but each request still carries and pays for its own full preamble; they amortize scheduling and rate-limit overhead, *not* the in-prompt instructional overhead this pattern targets. *Prompt caching* (e.g. DeepSeek) instead caches a repeated prompt prefix and discounts it, which *does* amortize the preamble's cost, but not its latency or the per-call overhead of many round-trips. Batching still earns its keep alongside either.

[7] https://developers.openai.com/api/docs/guides/batch
[8] https://platform.claude.com/docs/en/docs/build-with-claude/batch-processing

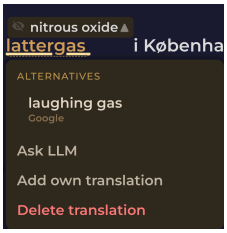

Fig. 2. In Zeeguu, the inline translation is the primary path; when the user wants a better rendering they escalate to an LLM on demand via the "Ask an AI" option.

3.2 Escalate to the LLM

3.2.1 Context. A feature can be produced two ways: a cheap, fast specialized tool (a translation API, a classical classifier) that is adequate for the common case, and a slower, costlier LLM that does better on the hard cases. Crucially, the cheap tool's shortfalls are *observable*: it either errors, or the user visibly rejects the result.

3.2.2 Example. In Zeeguu, translation APIs (from Google, Azure, and DeepL) serve as the primary translation engines. When a user indicates the translation is inadequate (by choosing *Ask an AI* from the alternatives menu), the system escalates to an LLM for a more nuanced, context-aware translation. This keeps costs low and speed high in the common case while providing higher, LLM-quality results when needed.

3.2.3 Problem. LLM quality is only worth its cost on the minority of requests the cheap tool handles poorly, and those cannot be told apart up front. How can the LLM be spent *only* where it pays off?

3.2.4 Forces.

- **Quality on the hard cases.** Specialized tools (translation APIs, NLP pipelines, classical classifiers) are faster, cheaper, and more deterministic than an LLM for well-defined tasks, but on a minority of inputs they fail or fall short, and there the LLM tends to do better. *(pushes toward escalating)*
- **Cost, latency, and usability.** Each escalation adds the LLM's cost and, because it runs after the cheap path, doubles the wait for that request; when the trigger is a user signal, it also costs UI surface and puts the routing burden on the user. *(pushes toward escalating rarely, and automatically where possible)*
- **The hard cases cannot be told apart up front.** The system cannot know which inputs the cheap tool will handle poorly before running it, so escalation must fire on an observable signal: the tool erroring, or the user rejecting the result.

3.2.5 Solution. Use the specialized tool as the primary path and escalate to the LLM only when the primary fails or the user signals dissatisfaction. The LLM receives the original input, and (where it helps) the specialized tool's output and the user's feedback alongside it, so the escalation refines rather than restarts.

3.2.6 Consequences.

- **Cheap in the common case.** The LLM runs only on failure or dissatisfaction, so the vast majority of requests never incur its cost or latency.

- **User-triggered escalation costs usability.** Unless it fires automatically on a detectable failure, the user must be given, and must notice and use, a way to signal dissatisfaction: extra UI surface, and the routing burden falls on them.
- **A miss doubles the wait.** When escalation happens the user has already waited for the cheap result first, so time-to-answer for those cases is the sum of both.

3.2.7 Known Uses. **FrugalGPT [2]** (Chen, Zaharia & Zou, 2023) escalates when a scorer rejects the cheap answer, and **RouteLLM [16]** (Ong et al., 2024) trains a confidence router that sends only the hard queries to the strong model. Both decide automatically from the model's own confidence, the *model cascade* variant of escalation discussed in the Notes below. The *external*-signal trigger this pattern centers on (the primary tool erroring, or the user rejecting the result) is less documented; Zeeguu is our instance of it.

3.2.8 Notes.

- *Applies broadly.* Beyond translation: topic classification, named entity recognition, or any task where a cheaper tool handles the common case and the LLM handles the long tail.
- *Distinct from Hybrid Classical+LLM Pipeline.* There a classical stage runs on every input as a recall gate and the LLM runs only on what it flags; here the cheap tool is the whole answer in the common case, and the LLM is reached only when it errors or the user rejects the result.
- *Escalation, not fallback.* Unlike a reliability fallback (where the secondary is an equal-or-lesser backup invoked when the primary fails), here the secondary is **more capable and more expensive**, invoked when the primary is not good enough. The movement is *up* in quality and cost, not *down* into degraded mode. That is why we name it escalation.
- *Relationship to the model cascade.* This is the human-/failure-triggered cousin of the **model cascade** in ML serving, where a cheap model runs first and a confidence threshold routes hard inputs to a larger model. The shared shape is *cheap tier first, expensive tier on demand*; the difference is the trigger. A cascade escalates automatically on the model's own low confidence, whereas this pattern escalates on external signals: the primary tool erroring, or the user explicitly declaring the result inadequate. A confidence-based cascade is thus one possible escalation policy; user dissatisfaction is another, and the two can be combined.
- *A black-box primary exposes no confidence to threshold on.* Zeeguu's translation APIs (Google, Azure, DeepL) return a translation but no calibrated per-request confidence, so the internal-confidence trigger a model cascade relies on is not even available here; the escalation signal has to come from outside the tool, its erroring or the user rejecting the result.

3.3 Hybrid Classical+LLM Pipeline

3.3.1 Context. A task can be served by a fast, deterministic classical tool (a dependency parser, a POS (part-of-speech) tagger, a rule extractor) that misses edge cases, and by an LLM that handles the edge cases but is too expensive to run on every input.

3.3.2 Example. Multi-word expression (MWE) detection (Section 2.1.4) runs Stanza [17] (a classical NLP library) first, as a cheap *high-recall* gate: it catches every possible candidate, tolerating false positives. If Stanza flags no candidate in a sentence, the LLM is never called. When it does flag one, an LLM re-analyzes the whole sentence and makes the

precision call, rejecting the false positives; its verdict is used even when it overrides Stanza and finds no expression. The LLM therefore runs on only the fraction of sentences that might contain an expression, rather than on every sentence.

3.3.3 Problem. How can both high recall and high precision be reached without paying LLM cost on every input?

3.3.4 Forces.

- **Recall at the gate.** The classical stage must flag every input the LLM might need to judge; anything it misses never reaches the LLM and is lost, so the gate is tuned for high recall. *(pushes toward a looser, more permissive gate)*
- **Cost of the LLM.** Every input the gate passes costs an LLM call, so a gate that flags too much erodes the saving that motivates the pattern. *(pushes toward a tighter gate)*
- **Precision needs the LLM.** The classical stage is fast but blunt; only the LLM can make the fine distinction and reject the false positives a permissive gate lets through. Neither stage alone achieves both high recall and high precision.

3.3.5 Solution. Run the cheap classical tool first, as a high-recall gate: invoke the LLM only when the classical stage fires, and skip it otherwise (the common case, and the main cost saving). When it fires, let the LLM make the precision decision. The two are not alternatives: the classical stage controls *when* the LLM runs; the LLM controls *what counts*.

3.3.6 Consequences.

- **The LLM runs only where it is needed.** It fires on the fraction of inputs the classical gate flags, so the common case costs nothing extra while the LLM still makes the precision call.
- **Two components instead of one.** The classical gate and the LLM stage are each built, tuned, and maintained separately, and the hand-off between them (what the gate flags, what the LLM is asked) has to stay correct as either evolves. That added complexity is usually justified by the saving from skipping the LLM on the common case.
- **The gate must have high recall.** A candidate the classical stage misses never reaches the LLM, so the recall of the cheap stage caps the recall, and thus the overall quality, of the whole pipeline.

3.3.7 Known Uses.

- **RankGPT [22]** (Sun et al., EMNLP 2023) uses BM25 (a classical keyword-ranking function) to retrieve ~100 candidates, then an LLM for listwise reranking: a classical high-recall generator with an LLM precision filter.
- The same *retrieve-then-rerank* shape, a high-recall first-stage search followed by a neural precision reranker (Nogueira and Cho [15]), is the standard two-stage architecture behind modern production search.

3.3.8 Notes.

- Close kin to *Escalate to the LLM* and to a model cascade: all three run a cheap step first and call the LLM selectively. The difference is the trigger. Escalate uses the cheap tool's answer and reaches for the LLM only when it is inadequate (a failure, or user dissatisfaction); here the cheap tool gates on detected difficulty (a flagged candidate) and the LLM's verdict replaces it, like a cascade, but the gating signal comes from a separate classical stage rather than the model's own confidence.

- *Enablers (not instances).* Frameworks such as spaCy-llm[9] (mixing LLM and rule-based components in one pipeline) and rerank products such as Cohere Rerank[10] make it easy to wire a classical stage to an LLM, but a library that provides the plumbing is the mechanism, not evidence of an in-app instance.

3.4 Anticipatory Precomputation

3.4.1 Context. A user-facing feature needs an LLM result, but the model takes seconds while the user expects a response in well under one, and *which* results a given user will need next is predictable from their behaviour.

3.4.2 Example. The vocabulary exercises a learner practices are built from the words they looked up while reading, each paired with a machine translation. At reading time an imperfect translation is low-stakes: the reader is only making sense of the text, and can pick a better option from the alternatives menu. But once the platform selects a word for the learner to *learn*, they will drill that word-translation pair over many days, so its correctness suddenly matters. A regular cron job looks ahead: it finds the words each learner is due to study next and validates them with an LLM, so that when the exercise module asks for new words, the vetted ones are ready straight away. If none are precomputed yet (the learner translated a few words and jumped straight into exercises), the validation runs in real time.

An even costlier instance: audio lessons (Section 2.1.7). Generating a personalized audio lesson is more expensive again: an LLM writes the lesson script (fed the learner's past lessons so a recurring topic, a "talking to a neighbour" lesson they have had before, comes back fresh rather than repeated), then text-to-speech synthesizes the audio, several seconds of work no learner should wait through. A nightly job precomputes the next lessons for recently active learners (prioritized by how recently they practiced), on the assumption that someone who has been studying will be back for the next one. When they return, the lesson is already waiting. Only a learner who switches to a new topic waits, briefly, while that day's first lesson on it is generated on demand.

3.4.3 Problem. How can an LLM-quality result reach the user's critical path without a wait for the model, when upcoming needs are often predictable?

3.4.4 Forces.

- **Latency.** Real-time users expect an answer in about 200ms, but an LLM can take several seconds depending on the prompt and deployment, so it cannot run on the critical path. *(pushes toward precomputing)*
- **Wasted spend on misses.** Precomputing spends tokens on results that may never be requested, so a poor predictor pays for nothing and still misses. *(pushes toward precomputing only high-probability needs)*
- **Predictability.** The pattern is available only where upcoming needs can be forecast from behaviour; the better the behaviour model, the more of the work can move off the critical path.

3.4.5 Solution. Anticipate likely user needs and precompute LLM results offline (e.g., via cron jobs), so results are available instantly when needed. The system designer should model user behaviour in order to predict their LLM needs. Keep an on-demand path for cold or mispredicted requests, so a miss degrades to a normal wait rather than a failure.

[9] https://github.com/explosion/spacy-llm
[10] https://docs.cohere.com/docs/rerank-overview

3.4.6 Consequences.

- **Zero latency at request time.** The LLM's cost and multi-second latency are paid entirely off the critical path; when the user acts, the result is already waiting.
- **Precomputing pays for guesses that miss.** It spends tokens on results that may never be requested, so the value depends on the accuracy of the behaviour model: a poor predictor wastes spend *and* still misses.
- **Needs a reliable "what" and "when" signal, plus a fallback.** It applies only where upcoming needs are predictable; cold or mispredicted requests still need an on-demand path. Composes with *Prompt Amortization* (offline results can be batched).

3.4.7 Known Uses.

- **Yelp**[11] precomputes LLM query-understanding responses for high-frequency ("head") search queries into a key/value store, "caching (pre-computing) high-end LLM responses for only head queries", reaching 95% of traffic for review-highlight expansions, so the expensive model never runs on the critical path.
- **Instacart**[12]'s Intent Engine serves high-frequency search queries from a precomputed cache of LLM outputs, leaving only ~2% of queries to a real-time model.
- **ColBERT [10]** (Khattab & Zaharia, SIGIR 2020) precomputes contextualized passage embeddings for the whole corpus offline, so query time runs only cheap late-interaction matching: the canonical "precompute the expensive representation ahead of time" (the pattern's shape applied to retrieval rather than generation).

4 Trusting LLM Output

An LLM's output is only probabilistically correct, and only probabilistically well-formed, yet it flows into code and data that assume it is neither wrong nor malformed. The patterns in this section guard that boundary: refusing to trust the shape of a response, catching content errors with a second, narrower check, recording how far each stored piece of output has been trusted. The unifying force is non-determinism: the same prompt can return a different, or malformed, answer, so correctness has to be enforced around the model rather than assumed from it.

4.1 Defensive Output Parsing

4.1.1 Context. An LLM is asked for output in a fixed shape (JSON, a delimited record), but format compliance is only probabilistic, and the parsed result feeds code on the critical path.

4.1.2 Example. Multi-word-expression (Section 2.1.4) detection asks the LLM for a JSON array (groups of word positions in a sentence) but refuses to blindly trust that it will get one.

- `_parse_response` first strips any markdown code fence (a triple-backtick code block the model often wraps the JSON in), tries `json.loads`, and on failure regex-extracts the last JSON array in the text (models tend to add a preamble), parses that, and distinguishes a legitimately empty result (`[]`) from a parse failure.
- If nothing parses, it logs the raw text and returns `[]` rather than raising.

[11] https://engineeringblog.yelp.com/2025/02/search-query-understanding-with-LLMs.html
[12] https://tech.instacart.com/building-the-intent-engine-how-instacart-is-revamping-query-understanding-with-llms-3ac8051ae7ac

- A separate `_validate_groups` step then checks the parsed shape (token indices in range) before anything downstream uses it.

The same layered try / extract / validate / fall-back appears in the translation validator, the simplification (Section 2.1.2) service, and the example generators.

4.1.3 Problem. How can a probabilistic formatting slip be kept from turning into a failed request?

4.1.4 Forces.

- **Format compliance is only probabilistic, and a naive parse fails the request.** The model can always wrap the output in a code fence, add a preamble, or truncate, and the parsed result feeds downstream code, so an unhandled slip turns into a failed request.
- **Strictness in the prompt is tempting but brittle.** Formatting failures invite ever-longer prompt instructions, which are hard to test and still only probabilistic. *(pushes toward prompt-stuffing)*
- **Strictness in parsing code is deterministic and testable**, at the cost of code to write and maintain. *(pushes toward defensive parsing)*

4.1.5 Solution. Do not trust the raw output's shape.

Parse in layers: strip known wrappers, attempt a lenient parse, extract the expected structure from the surrounding text if that fails, validate the parsed shape (types, ranges, required fields), and then degrade gracefully (a default, a skip, a retry, or the next provider in a fallback chain of LLM vendors) instead of raising.

Keep the strictness in code, where it is deterministic and testable, rather than in ever-longer prompt instructions.

4.1.6 Consequences.

- A malformed response degrades a single result instead of failing the request: the layered parse recovers what it can, and a clean fallback (a default, a skip, a retry, the next provider) covers the rest.
- The parse layer is code to write and maintain, and it has to be kept in step with the prompt and the provider as the output format drifts.
- Provider "JSON mode" or function-calling reduces malformed output but does not remove the need to validate the shape before use.

4.1.7 Known Uses.

- **G-Research**[13]'s code-review tool treats LLM output as "unverified input": it normalises responses by "removing wrappers before parsing" (some providers return bare JSON, some wrap it in markdown fences), validates every finding against an authoritative rule index (invalid findings are "rejected outright"), detects truncation via the `length` finish reason and retries with a lower cap, and sends structurally-invalid output back to the model for one repair pass.
- **Honeycomb**[14]'s Query Assistant parses the LLM output, "correct[s] it (if it's correctable)," validates it, and instruments parse vs. validation errors separately before running the query.

[13] https://www.gresearch.com/news/building-a-code-review-tool-the-llm-patterns-that-actually-work/
[14] https://www.honeycomb.io/blog/hard-stuff-nobody-talks-about-llm

4.1.8 Notes.

- Composes with a one-shot retry and with a provider fallback chain (a parse failure can trigger the next provider).
- Broader than repairing a specific, known formatting defect: this pattern is the stance of not trusting the structure at all.
- *Prior art: error-tolerant parsing.* Recovering the expected structure and skipping the surrounding text is the LLM-output analogue of island grammars [14] (parse the fragments of interest, skip the rest) and other lenient, error-tolerant parsing long used to pull structure out of irregular input.
- *Enablers (not instances).* Validation/repair is widely productized, Instructor[15] (Pydantic + auto-retry), LangChain[16] `OutputFixingParser`, and provider structured-output[17] modes (which still require handling truncation and refusals), but a library that *provides* validation is the mechanism, not evidence of an in-app stance.

4.2 LLM-Checking-LLM

4.2.1 Context. An LLM generates content that will be used or stored, and its output is sometimes wrong in ways a targeted check could catch. Verifying a specific property (grammaticality, factual match, difficulty level) is a narrower task than the open-ended generation that produced it.

4.2.2 Example. The vocabulary exercises need example sentences for each word, which an LLM generates at the learner's level, an open-ended task (the sentence must be natural, level-appropriate, and actually use the word). A generated sentence can still be wrong in a specific way: it may use the word in a *different* sense than the one being taught. For the Danish *virker* (translated as *seem*), a generated sentence might use *virker* in its other sense, *work/function*. A second, batched LLM call then asks one narrow question of each sentence, whether it uses the word in the intended meaning (Section 2.1.5), and drops the ones that fail. Verifying that single property is far narrower than writing a good sentence from scratch.

4.2.3 Problem. How can an unreliable generator's mistakes be caught, when a second generator would be just as unreliable?

4.2.4 Forces.

- **Verification is narrower than generation.** Checking one property (is it grammatical? does it use the intended meaning?) has a small answer space and a clear criterion, so a focused checker is more reliable on that property than the open-ended generation was: the *generation-discrimination gap* measured by Saunders et al. [19]. *(pushes toward adding a check)*
- **The checker is itself an LLM.** It can return its own false verdicts, and it costs a full extra call in tokens and latency. *(pushes toward checking only where the asymmetry is large, or where a classical check exists)*
- **The mistakes must be checkable in isolation.** The pattern helps only for errors a targeted, differently-prompted call can catch, not for failures that need the whole generation redone.

[15] https://python.useinstructor.com/
[16] https://python.langchain.com/api_reference/langchain/output_parsers/langchain.output_parsers.fix.OutputFixingParser.html
[17] https://developers.openai.com/api/docs/guides/structured-outputs

4.2.5 Solution. Use one LLM call to generate a result, then a separate, differently-prompted LLM call to check one specific property of it. This escapes the paradox for two reasons. First, verifying one property (is it grammatical? does it use the intended meaning?) has a small, well-defined answer space and a clear success criterion, so an LLM does it more reliably than the open-ended generation that produced the output. Second, because the checker is prompted differently and asked a different question, its errors are largely *decorrelated* from the generator's rather than shared, so it does not simply repeat the generator's mistakes.

4.2.6 Consequences.

- **A focused check is more reliable than the generation.** Verifying one property is easier than producing the whole output, so the second call catches errors the first introduced, for the price of one extra call.
- **It narrows the error rate, it does not remove it.** The checker is itself an LLM and can return its own false verdicts, and it adds cost and latency.
- **It pays off only when checking is genuinely narrower than generating.** A check as open-ended as the generation buys little. The verdict composes with *LLM Content Validation Tracking* (record it) and *Hybrid Classical+LLM Pipeline* (a classical check is cheaper still, where one exists).

4.2.7 Known Uses.

- *Documented in the literature.* **LLM-as-a-judge [26]** (Zheng et al., NeurIPS 2023) uses a separate strong LLM to score another model's open-ended outputs, now a standard evaluation technique.
- **Self-Refine [13]** (Madaan et al., NeurIPS 2023) has the model critique and refine *its own* output in a separate pass: a same-model variant of the idea, where this pattern instead uses a differently-prompted call.
- **G-Eval [12]** (Liu et al., 2023), shipped in eval frameworks such as **DeepEval**[18], uses a separate chain-of-thought (step-by-step reasoning) judge call to grade generated output.

4.2.8 Notes.

- Where *Defensive Output Parsing* guards that the output is well-*formed*, this guards that a well-formed output is *correct*.
- Distinct from ensemble methods and chain-of-thought: an ensemble averages several generations and chain-of-thought elaborates one, whereas this pattern spends the second call on the cheaper *verification* task instead of more generation.
- *Why the asymmetry holds, and where it stops.* Spending a call on verification rather than generation echoes the intuition behind NP, the class of problems where a proposed answer can be checked quickly even when finding one may be hard: confirming a candidate can be far easier than producing it. The gain is empirical here, not guaranteed, and it depends on the check being a narrow, differently-prompted property. It does not extend to open-ended *intrinsic self-correction*, a model revising its own reasoning with no new signal, which shows no such gain and can even degrade accuracy (as Huang et al. [9] and Stechly et al. [21] find). That is why the check here asks a *different* question through a *separate* call, rather than asking the generator to reconsider.
- *This check may be transient.* It exists because today's models are imprecise enough to need an external verifier. As models improve, or as reliable self-verification moves inside the model itself, the need for a separate checking call may shrink: the pattern answers the current generation's reliability, not a permanent architectural truth.

[18] https://deepeval.com/docs/metrics-llm-evals

4.3 LLM Content Validation Tracking

4.3.1 Context. LLM-generated content is written into persistent storage alongside human-verified data, and downstream features and users will read it. Some of it has been checked, most has not, and once stored the two look identical.

4.3.2 Example. A meaning (Section 2.1.5) (a word paired with a translation) can exist at several trust levels: generated by Google Translate (unverified), verified by an *LLM-Checking-LLM* pass (auto-verified), implicitly accepted (a learner drilled it in an exercise without flagging it as wrong), or explicitly corrected by a trusted user, such as a teacher. Each level carries different confidence, and the system can make different decisions depending on the validation state: for example, only including explicitly confirmed translations in exercises, while using auto-verified ones for reading assistance where the stakes are lower.

4.3.3 Problem. How can trusted and untrusted LLM-generated data be told apart after it lands in the database, so each is used according to its confidence?

4.3.4 Forces.

- **Unmarked data is silently trusted.** Once stored, LLM output is indistinguishable from human-verified data, so without a marker downstream features treat unverified output as ground truth. *(pushes toward tracking trust at all)*
- **Different consumers can tolerate different confidence.** Reading assistance is fine with an unverified translation; a word promoted into drills should be confirmed first. Serving both well means distinguishing levels of trust, not just verified from unverified. *(pushes toward a richer trust spectrum)*
- **Every level is state to maintain.** Each level is another transition to model, set on write and update on every validation event, so too fine a spectrum is maintenance burden and false precision. *(pushes toward the coarsest spectrum that still supports the decisions)*

4.3.5 Solution. Maintain an explicit, queryable validation state for all LLM-generated content in the data model. Never let LLM-generated content silently become trusted data. The validation state may be a simple flag, but more often it is a spectrum or state machine reflecting different levels of trust (e.g., unverified → auto-verified → implicitly accepted → explicitly confirmed by a trusted user; alternatively, one could track the number of users that have validated a given content).

4.3.6 Consequences.

- **Each consumer can gate on trust.** Exercises use only confirmed pairs, reading assistance can fall back to auto-verified ones, and unverified output never silently becomes ground truth.
- **The data model carries an extra state to maintain.** It must be set on every write and updated on every validation event, and the right granularity (a flag, a spectrum, or an agreement threshold: N independent users confirming before it counts as trusted) is a domain judgment.
- **It pairs with provenance to describe any stored artifact.** Where *LLM Output Provenance* records how an artifact was made, this records whether it has been confirmed; together they answer both questions about a stored artifact.

4.3.7 Known Uses.

- **Argilla**[19], an open-source platform for annotating and reviewing model-generated data, gives every record an explicit status: a model suggestion starts as *pending*, becomes a *draft* and then *submitted* as someone works on it, and ends up *validated* or *discarded*. A suggestion is never treated as correct until a human has moved it to *validated*.
- **Label Studio**[20], another annotation tool, flags each item as ground truth or not and records whether it has been reviewed, so human-verified "gold" data stays clearly separate from unreviewed model predictions.
- **Snorkel [18]** (Ratner et al., VLDB 2017) attaches probabilistic confidence to programmatically generated labels rather than treating them as ground truth: a confidence spectrum rather than a discrete state machine.

Together the three bracket the granularity this pattern turns on: a multi-state lifecycle (Argilla), a binary gold/not-gold flag (Label Studio), and a probabilistic score (Snorkel).

Note. The examples above are data-labeling platforms; a documented, in-product *LLM-output* trust-state lifecycle (as opposed to human-annotation state) remains thinly evidenced: one reason we keep this among the less-settled patterns.

4.3.8 Notes.

- This pattern complements *LLM Output Provenance*. Together they answer two essential questions about any piece of LLM-generated data: how was it produced? and has anyone confirmed it's correct? The right granularity of validation tracking depends on the domain: some systems may need binary (verified/not), others may need multiple validators with agreement thresholds, and others, like Zeeguu, benefit from a trust spectrum that reflects different forms of implicit and explicit user feedback.
- *Implicit signals are weaker than explicit ones.* A word drilled in exercises without ever being flagged is a positive signal, but a weak one, which is why it sits below a correction from a trusted user in the trust order.

5 Managing Change Over Time

An LLM integration is not static: prompts and models improve, vendors retire dated snapshots on their own schedule, and the LLM's own role in a feature shifts as the system matures. The patterns in this section manage that change: stamping stored output with the model and prompt that made it so the stale can be found and regenerated, retiring stale artifacts without breaking the past, and treating the LLM as a temporary stand-in for a cheaper component still to be built. The unifying force is a fast-moving, vendor-controlled substrate sitting under long-lived data.

5.1 LLM Output Provenance

5.1.1 Context. LLM-generated artifacts (example sentences, summaries, labels) are written to persistent storage and reused for a long time, while the models and prompts that produce them keep improving. The prompt changes more often than the model, and can matter as much to output quality, sometimes more.

[19] https://docs.argilla.io/latest/how_to_guides/annotate/
[20] https://docs.humansignal.com/guide/ground_truths

5.1.2 Example. When the system generates example sentences for a word, it stamps each stored sentence with a `created_by` value naming the model and prompt version that produced it (for example, `claude-opus / examples-v3`). When the example-generation prompt is improved to `v4`, the stale sentences are exactly those still stamped `v3`, so a single query finds them and they are regenerated, without touching the rest of the store.

5.1.3 Problem. When a prompt or model improves, how can exactly the stale artifacts be found and regenerated, without reprocessing the entire store?

5.1.4 Forces.

- **Selective regeneration needs to know how each artifact was made.** Without a record of the model and prompt behind a stored artifact, applying an improved prompt means reprocessing everything. *(pushes toward stamping provenance)*
- **A stamp is only useful if it stays truthful.** Provenance has to be written on every artifact and kept in lockstep with the code: a prompt edited in place without bumping its version leaves the stamp naming the wrong one, silently lying, which is worse than no stamp at all. Keeping it honest is an ongoing discipline, not a one-off write. *(pushes toward stamping precisely what drives regeneration, and versioning it rigorously)*
- **The prompt is the higher-churn axis**: it changes more often than the model and can change the output more, so the stamp must capture the prompt version, not just the model.

5.1.5 Solution. Store the full provenance tuple alongside every LLM-generated artifact: **model version, prompt version, generated output, timestamp**. This turns regeneration into a targeted query, as broad as *"re-run everything produced by prompt v2 with the improved v3"* or as narrow as a single *(model, prompt)* pair, so when a prompt does well on one model and poorly on another, only that combination is redone.

5.1.6 Consequences.

- **Selective regeneration becomes a query.** Re-run everything a given prompt or model produced and leave the rest, and the same record doubles as a quality-audit trail.
- **The guarantee lasts only as long as the discipline.** Selective regeneration is trustworthy only while every write stamps accurately; once a stamp drifts from what actually produced the artifact, queries built on it quietly return the wrong rows.
- **One identifier, shared with selection and validation.** The stamped model identifier and the one used to select the model at call time should be the same central constant, kept in one place, and provenance pairs with *LLM Content Validation Tracking*: how an artifact was made, and whether it has been confirmed.

5.1.7 Known Uses.

- *Better attested in tooling than in production self-reports.* The capability is productized: MLflow Prompt Registry[21] and LangSmith[22] version prompts and record which prompt/model produced each output, confirming the pattern's core claim that the *prompt* deserves versioning as much as the model, but these are tools, not documented in-app deployments.

[21] https://mlflow.org/docs/latest/genai/prompt-registry/
[22] https://docs.langchain.com/langsmith/prompt-engineering-concepts

- *Adjacent production pipelines.* DoorDash[23] stores versioned LLM-generated profiles and "treat[s] prompts as code"; Etsy[24] stores Pydantic-validated LLM attribute extractions over 100M+ listings. Both store LLM artifacts at scale and version prompts, but neither publicly describes stamping *each artifact* with its prompt version to drive *selective* regeneration: the load-bearing mechanic here. We did not find a first-hand account of the full pattern, so Zeeguu is our instance.

5.1.8 Notes.

- The key insight is that the prompt is at least as important to version as the model: a prompt change can completely alter output format, quality, or behaviour even with the same model.
- This is also critical for *Rent, Then Build*: when accumulating LLM-generated labels as training data for a classical replacement, provenance tracking lets one exclude data produced by a prompt version that was later found to be noisy or biased.
- *Implicit provenance* keeps model names and prompt versions as constants in code and, when the origin of a row is needed, correlates its `created_at` timestamp with git history to find which model/prompt was deployed then. This works only where a single model/prompt is active at a time. Once more than one is live at once (A/B tests, multi-model routing, or the second model of an *Escalate to the LLM* path), the timestamp no longer identifies a unique version and provenance must be stamped explicitly. Explicit tracking also makes analysis faster and keeps the data self-describing.
- *Gateways log calls, not artifacts.* An LLM gateway records each request and response, but selective regeneration needs the provenance to live on the stored artifact and reach the database, where a query can find the stale rows; request logs alone do not drive it.

5.2 Soft Invalidation of LLM Artifacts

5.2.1 Context. An LLM-generated artifact has been stored and is reused as a cache, and it is also referenced from user-visible history. Then the prompt or model that produced it improves, so the stored version is now known to be suboptimal while old references still point at it.

5.2.2 Example. When the prompt that generates audio lesson (Section 2.1.7) scripts was improved, the ~900 stored `audio_lesson_meaning` rows (each caching one vocabulary item's generated audio) produced under the previous prompt were neither regenerated eagerly nor deleted: a learner who had already listened to one should not have its content change underneath them. Instead, each affected row received a `deprecated_at` timestamp, and the cache-lookup helper (`AudioLessonMeaning.find()`) was gated to skip deprecated rows. New daily lessons request a fresh row and trigger regeneration under the new prompt; existing daily lessons that already reference a deprecated row keep playing their old audio without breaking.

5.2.3 Problem. When the generator improves, how can stored artifacts be refreshed without either paying to regenerate everything up front or breaking the past references that still point at the old ones?

5.2.4 Forces. When a prompt or model improves, the obvious responses each have a serious drawback:

[23] https://careersatdoordash.com/blog/doordash-profile-generation-llms-understanding-consumers-merchants-and-items/
[24] https://www.etsy.com/codeascraft/understanding-etsyas-vast-inventory-with-llms

- *Regenerate everything eagerly*: expensive, floods generation queues if affected rows number in the thousands, and pays for content that may never be re-requested.
- *Delete the stale rows*: breaks any downstream object that references them by id (history, analytics, user-visible past sessions).
- *Leave the stale rows in place and accept future reuse*: silently propagates the old, known-suboptimal quality.

None of these are good defaults for production systems where LLM-generated artifacts are referenced from user-visible history and are also targets for reuse.

5.2.5 Solution. Mark stale rows as deprecated rather than mutating or removing them. Gate the cache-lookup / reuse path to skip deprecated rows, forcing fresh generation on next demand. Existing references to a deprecated row remain valid (the row keeps its content for historical playback), but no new consumer picks it up. For this to preserve history, key each stored artifact by its own row id, not by the source it describes, so a regenerated row produces a new file rather than overwriting the deprecated one (see the note). Regeneration cost is paid lazily, amortized over normal access patterns, and only for content that is actually requested again.

5.2.6 Consequences.

- **Cost is lazy and history stays intact.** Regeneration is paid only for content requested again, and old references keep resolving to their original artifact, so user-visible history does not break.
- **Old versions linger until next demand.** A replay hears the old quality until something triggers regeneration, and the deprecation flag has to reach every downstream cache to be effective.

5.2.7 Known Uses.

- **`stale-while-revalidate`**[25] (IETF RFC 5861) is the same mechanic in HTTP caching: keep a stale entry usable and revalidate it lazily on next demand rather than eagerly regenerating.
- The **soft-delete / tombstone**[26] database idiom marks a row with a timestamp and gates every read (`WHERE deleted_at IS NULL`), so the row stays intact for existing references but drops out of the active path: structurally identical to a `deprecated_at` gate.

Note that both the above are *analogues and not exact instances.* Both capture the mechanism, but we did not find a documented LLM system that combines mark-deprecated + gate-reuse + **keep-old-rows-resolvable-for-user-history** + lazy regeneration; the history-preservation facet appears novel, so we present it as our own contribution, grounded in Zeeguu.

5.2.8 Notes.

- **The mechanism is soft-delete; the trigger is what makes it LLM-specific.** A soft-delete retires a record someone chose to remove. Here nothing is deleted and nothing is wrong: the row is retired from reuse only because the prompt or model that produced it improved, so its once-good output is now merely suboptimal. Unlike a cached database row, a stored LLM artifact goes stale on its own as its generator gets better. In Zeeguu the trigger was a prompt rewrite, the common case, since prompts change more often than models.
- This pattern is forward-only: it gates *reuse*, not *playback*. A user replaying an old lesson hears the old (lower-quality) version. That is usually preferable to a silent content swap mid-history.

[25] https://datatracker.ietf.org/doc/html/rfc5861
[26] https://brandur.org/soft-deletion

- Composes naturally with *LLM Output Provenance*: provenance answers "which rows are stale?", soft invalidation answers "what do I do with them once I know?".
- **Name the stored file after the row, not after what it describes.** The history guarantee needs each regeneration to produce a *new* file rather than overwrite the old one. In Zeeguu this broke at first: each lesson's audio was stored under its `meaning_id` (the vocabulary item the lesson teaches), which stays the same when the lesson is regenerated. So regenerating a deprecated lesson wrote the new audio to the same path and silently replaced the recording that old daily lessons still played. The fix was to key each file by the row's own id (`audio_lesson_meaning.id`), so every generation gets its own file and the deprecated one survives for playback.

5.3 Rent, Then Build

5.3.1 Context. A feature needs to work in production now, but the efficient, dedicated version of it (a fine-tuned or classical model) needs training data or engineering that does not exist yet. A general-purpose LLM can do the task immediately, if expensively.

5.3.2 Example. Zeeguu estimates the difficulty level of every article with an LLM, an expensive call it makes constantly. This is also the *bootstrapping* case: those LLM-generated difficulty labels are accumulating as the training set for a cheaper classical classifier that will take the task over once enough have been gathered. The LLM ships the feature and earns its keep today, while quietly producing the data that will one day retire it.

A plainer instance without the bootstrapping twist: article topic classification runs on an LLM now, to be replaced by a dedicated topic-detection model once the taxonomy of available topics settles.

5.3.3 Problem. How can a feature ship and start earning its keep before the cheaper, dedicated version that will eventually run it has been built?

5.3.4 Forces.

- **Shipping now captures the feature and its usage immediately.** A general-purpose LLM can do the task the day it is needed, before any dedicated version exists. *(pushes toward renting)*
- **Running the LLM indefinitely is costly and cedes control.** The feature keeps paying per call, and a core capability stays rented from a vendor who prices and deprecates on their own schedule. *(pushes toward building a replacement)*
- **The replacement cannot be built yet.** A fine-tuned or classical version needs training data or engineering that does not exist at the outset; only running the LLM produces it.

5.3.5 Solution. Use the LLM to perform a task in production while building a more efficient replacement. The arc is rent, then build: the LLM's general capability is *rented* (paid per call, costly but available immediately) to ship the feature now, while a cheaper, dedicated implementation is *built* to eventually own the task. The rented LLM is convincing to users and good for early beta-testing and feedback, but intended to be temporary.

Bootstrapping variant: In the strongest form, the LLM *generates the training data for its own replacement* (as with the difficulty labels above): the rented capability directly produces the labeled data the dedicated successor is trained on, so running the stand-in funds and enables the build.

5.3.6 Consequences.

- **The feature is live from day one.** It gathers real usage and feedback immediately, with the LLM standing in for a component that does not exist yet.
- **Running the stand-in funds its successor.** The LLM's inputs and outputs accumulate as the labeled data the dedicated replacement needs, so the temporary solution also produces the training set for the permanent one (the bootstrapping variant).
- **The stand-in is expensive, and "temporary" can stick.** Until the replacement ships, the feature runs at LLM cost and latency, the replacement is a second system to build and validate, and the intended-temporary LLM can quietly become permanent.

5.3.7 Known Uses.

- **OpenAI Model Distillation**[27] (Stored Completions) captures a large model's production input–output pairs and fine-tunes a smaller model as a drop-in replacement: the pattern shipped as a product feature.
- **Self-Instruct / Alpaca [23]** (Wang et al.; Taori et al., 2023) use a strong LLM to generate the instruction data that fine-tunes a small replacement: the bootstrapping variant.
- **Distilling Step-by-Step [8]** (Hsieh et al., ACL Findings 2023) trains task-specific models up to 700× smaller from LLM-generated rationales.

These are distillation and self-instruct methods from the literature; we did not find a first-hand account of a production feature completing the full rent-then-build arc, and Zeeguu's own replacement is still in progress.

5.3.8 Notes.

- This pattern has a lifecycle relationship with *Escalate to the LLM*: a system may start with the LLM as primary (renting it), migrate to a specialized tool as primary, and then keep the LLM as the escalation path, completing a full cycle from LLM-first to LLM-on-demand.
- *Depends on LLM Output Provenance* for the bootstrapping variant: because the replacement trains on the rented LLM's own outputs, those labels must be filterable by the prompt and model version that produced them, so labels from a prompt version later found noisy or biased can be excluded from the training set.
- *Composes with LLM Content Validation Tracking* in the bootstrapping variant: gating the training set to validated outputs (confirmed, not merely generated) lets the dedicated successor learn from checked labels rather than the stand-in's unverified guesses.

6 What Makes These Patterns LLM-Specific?

Some of these patterns echo general distributed systems wisdom: batching (as in *Prompt Amortization*), fallback (as in *Escalate to the LLM*), and the redundant dispatch of Zeeguu's parallel translation providers (not catalogued here as a pattern of its own). What makes them distinctly relevant to LLM integration is the combination of forces that arise when an LLM's properties meet the demands of a live system:

- **Cost structure**: per-token pricing with high fixed prompt overhead, unlike flat-rate API calls (drives *Prompt Amortization*, *Escalate to the LLM*, *Anticipatory Precomputation*).
- **Non-determinism**: the same input can yield a different, or malformed, output, so correctness must be enforced around the model (drives *Defensive Output Parsing*, *LLM-Checking-LLM*, *LLM Content Validation Tracking*).

[27] https://openai.com/index/api-model-distillation/

- **Asymmetry between generation and verification**: checking one property is easier than producing the whole output (drives *LLM-Checking-LLM*).
- **General-purpose capability**: the same component can serve as prototype, primary, or fallback (drives *Rent, Then Build* and *Escalate to the LLM*).
- **A rapidly evolving, vendor-controlled substrate**: models and prompts improve and are deprecated on the vendor's schedule, underneath long-lived data (drives *LLM Output Provenance* and *Soft Invalidation of LLM Artifacts*).
- **Quality-cost-latency tradeoff space**: uniquely wide compared to traditional APIs, and what the efficiency patterns navigate.

7 Related Work

To our knowledge, no peer-reviewed work presents a catalogue of architectural patterns for integrating LLMs as components into existing production systems, grounded in real deployment experience and described using the standard pattern format (context, forces, solution, consequences). The contribution is the catalogue itself, organised by three themes: using the LLM efficiently, trusting its output, and managing change over time. Some patterns apply established mechanisms (batching, precomputation, recall gates, soft-delete, distillation) to the distinctive forces of LLM integration; others, notably *LLM Output Provenance*, *LLM Content Validation Tracking*, and the external-signal trigger of *Escalate to the LLM*, appear to be new. We flag each honestly as one or the other throughout.

Several practitioner-oriented resources discuss patterns for building LLM-based systems, but they operate at a different level of abstraction and lack the grounding in a specific production system that we aim to provide.

7.1 Practitioner resources

Eugene Yan's **Patterns for Building LLM-based Systems & Products [24]** (2023, blog post) identifies seven patterns: Evals, RAG, Fine-tuning, Caching, Guardrails, Defensive UX, and Collecting User Feedback. These address the question "how do I build an LLM product?" They describe the overall stack for LLM-native applications. Our patterns address a different question: "I have an existing system, and I want to add LLM capabilities as a component: how do I manage cost, quality, latency, and lifecycle?" There is some overlap on caching/pre-computation, but our treatment focuses on prompt amortization and user-need anticipation rather than semantic similarity caching. Yan's work is not peer-reviewed.

ThoughtWorks' **Emerging Patterns in Building GenAI Products [5]** (Fowler et al.) and "Engineering Practices for LLM Applications" focus on operational concerns: testing, evaluation, guardrails, and RAG pipelines. These are primarily about LLMOps rather than the architectural decisions for embedding LLMs as components within an existing application.

Andreessen Horowitz's **Emerging Architectures for LLM Applications [1]** (2023) provides a reference architecture for the LLM infrastructure stack (embedding pipelines, vector databases, orchestration layers, agents). Again, this targets LLM-native products rather than LLM integration into existing systems.

Books. "**LLM Design Patterns**" (Huang, Packt, 2024) and "LLMs in Enterprise" (Menshawy & Fahmy, Packt, 2025) cover model-level patterns (fine-tuning, quantization, inference optimization, RAG) and enterprise deployment concerns. They do not address application-level integration patterns such as the lifecycle management (Rent, Then Build → specialized tool → Escalate to the LLM), prompt amortization, or LLM output provenance that we identify.

7.2 Academic surveys

There is a large body of work on **using LLMs *for* software engineering tasks**: code generation, bug repair, testing, requirements engineering (see surveys by Fan et al., 2023 [4]; Zhang et al., 2024 [25]). However, these focus on LLMs as tools for developers, not on the engineering challenges of integrating LLMs as runtime components within production software.

A recent **systematic literature review on software architecture and LLMs** (Schmid et al., 2025 [20]) found only 18 relevant papers and noted that LLM-based software design remains an open research direction. None of the surveyed academic work addresses the specific architectural patterns for managing cost, quality, latency, and lifecycle when LLMs serve as components in existing applications.

LLM self-verification. For the *LLM-Checking-LLM* pattern, there is relevant work on **LLM self-verification**. Gero et al. (2023) [6] demonstrated that self-verification improves clinical information extraction accuracy, explicitly building on the asymmetry between verification and generation. However, Stechly et al. (2024) [21] showed that self-critique fails for formal reasoning tasks, finding significant performance collapse with self-verification but gains with external verification. Our pattern differs from both in that it uses separate, differently-prompted LLM calls for generation and verification of different properties (e.g., text simplification followed by grammatical correction), rather than asking an LLM to verify its own reasoning.

8 Limitations and Future Work

The patterns in this catalogue have been extracted from a single production system, Zeeguu, in the language-learning domain. We have argued throughout that these forces arise from the LLM's own properties (per-token cost, multi-second latency, non-determinism, imprecision, a rapidly shifting provider landscape, and general-purpose capability) meeting the demands of a live application, neither of which is specific to language learning, and that the patterns should therefore generalise to other interactive applications that surface LLM-generated text to end users. Whether this is borne out in practice is an empirical question we cannot fully answer from a single case study. The most direct way to validate or refute the catalogue is to gather complementary examples (and counter-examples) from practitioners working in other domains. A PLoP writers' workshop is one venue for exactly this kind of community refinement; mining open-source repositories for additional instances of these patterns (and patterns we missed) is a natural follow-up.

A second limitation is that the catalogue reports patterns we have found useful but does not yet quantify their impact systematically. Cost and latency improvements are reported informally for some patterns; a more rigorous deployment-level evaluation (measuring, for example, how much of Zeeguu's LLM bill is attributable to which patterns, or how much user-perceived latency racing providers removes) is future work.

A further direction is to treat the catalogue as the seed of a *pattern language* rather than a flat list. The patterns are not independent: they compose (pre-computation feeds *Prompt Amortization*; *LLM Output Provenance* flags which artifacts *Soft Invalidation of LLM Artifacts* should retire) and they evolve over a system's lifetime (an LLM may begin as a *Rent, Then Build* stand-in, hand off to a specialized tool, and remain as the *Escalate to the LLM* fallback). Documenting these interactions, sequences, and tensions, so a designer can navigate from one pattern to the next, is a contribution beyond the individual patterns.

These choices are not made once. A system's pattern mix is actively revised as it scales and the provider landscape shifts: Zeeguu began by making single LLM calls and only recently added parallel, raced ones, and may later drop

that approach or fold it together with per-user budgeting as usage grows. A longitudinal account of how and why a system's chosen patterns change over time would deepen the lifecycle dimension that this single-snapshot catalogue can only sketch.

Finally, we expect the catalogue to be incomplete[28]: further patterns will likely surface only through engagement with practitioners working at different scales, in different domains, and with different LLM providers.

9 Conclusion

This paper has presented a catalogue of architectural patterns for integrating LLMs as components into existing user-facing applications: a setting that has received much less attention than the design of LLM-native products, despite arguably affecting a larger share of working software engineers. The patterns address recurring concerns around cost, latency, lifecycle, data management, and quality assurance, and were drawn from real production experience on the Zeeguu platform. They are intended as a starting point for community refinement rather than a closed taxonomy: practitioners working on LLM integration in other domains are warmly invited to validate, refute, extend, or replace individual patterns, and to contribute new ones the catalogue does not yet contain.

10 Acknowledgments

We thank Souhaila, Oscar, and Diomidis for feedback and discussions on earlier versions of this work. We are also grateful to the participants of the EuroPLoP focus group, whose discussion sharpened several patterns, from their naming to their relationship with both older, non-LLM patterns and the emerging LLM-gateway infrastructure.

References

[1] Andreessen Horowitz. 2023. Emerging Architectures for LLM Applications. https://a16z.com/emerging-architectures-for-llm-applications/

[2] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. *Transactions on Machine Learning Research (TMLR)* (2023). https://arxiv.org/abs/2305.05176

[3] Zhoujun Cheng, Jungo Kasai, and Tao Yu. 2023. Batch Prompting: Efficient Inference with Large Language Model APIs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track (EMNLP)*. https://arxiv.org/abs/2301.08721

[4] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. https://arxiv.org/abs/2310.03533

[5] Martin Fowler et al. 2024. Emerging Patterns in Building GenAI Products. https://martinfowler.com/articles/gen-ai-patterns/

[6] Zelalem Gero, Chandan Singh, Hao Cheng, Tristan Naumann, Michel Galley, Jianfeng Gao, and Hoifung Poon. 2023. Self-Verification Improves Few-Shot Clinical Information Extraction. In *ICML 3rd Workshop on Interpretable Machine Learning in Healthcare (IMLH)*. https://arxiv.org/abs/2306.00024

[7] Alex Gilmore. 2007. Authentic materials and authenticity in foreign language learning. *Language Teaching* 40, 2 (2007), 97–118. https://doi.org/10.1017/S0261444807004144

[8] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling Step-by-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes. In *Findings of the Association for Computational Linguistics (ACL)*. https://arxiv.org/abs/2305.02301

[9] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large Language Models Cannot Self-Correct Reasoning Yet. https://arxiv.org/abs/2310.01798

[10] Omar Khattab and Matei Zaharia. 2020. ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. https://arxiv.org/abs/2004.12832

[11] Stephen D. Krashen. 1982. *Principles and Practice in Second Language Acquisition*. Pergamon Press. https://sdkrashen.com/content/books/principles_and_practice.pdf

[12] Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://arxiv.org/abs/2303.16634

[28] The patterns in this paper are a selection from a larger, evolving catalogue maintained online at https://mircealungu.github.io/llm-integration-patterns/

[13] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*. https://arxiv.org/abs/2303.17651

[14] Leon Moonen. 2001. Generating Robust Parsers Using Island Grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE)*. 13–22. https://doi.org/10.1109/WCRE.2001.957806

[15] Rodrigo Nogueira and Kyunghyun Cho. 2019. Passage Re-ranking with BERT. https://arxiv.org/abs/1901.04085

[16] Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. 2024. RouteLLM: Learning to Route LLMs with Preference Data. *arXiv preprint arXiv:2406.18665* (2024). https://arxiv.org/abs/2406.18665

[17] Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A Python Natural Language Processing Toolkit for Many Human Languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL): System Demonstrations*. https://arxiv.org/abs/2003.07082

[18] Alexander Ratner, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proceedings of the VLDB Endowment* 11, 3 (2017). https://arxiv.org/abs/1711.10160

[19] William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. 2022. Self-critiquing models for assisting human evaluators. https://arxiv.org/abs/2206.05802

[20] Larissa Schmid et al. 2025. Software Architecture Meets LLMs: A Systematic Literature Review. (2025). https://arxiv.org/abs/2505.16697

[21] Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. 2024. On the Self-Verification Limitations of Large Language Models on Reasoning and Planning Tasks. https://arxiv.org/abs/2402.08115

[22] Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. 2023. Is ChatGPT Good at Search? Investigating Large Language Models as Re-Ranking Agents. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://arxiv.org/abs/2304.09542

[23] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA Model. https://github.com/tatsu-lab/stanford_alpaca. https://github.com/tatsu-lab/stanford_alpaca

[24] Eugene Yan. 2023. Patterns for Building LLM-based Systems and Products. https://eugeneyan.com/writing/llm-patterns/

[25] Quanjun Zhang, Chunrong Fang, Yang Xie, Yuxiang Zhang, Yang Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2024. A Survey on Large Language Models for Software Engineering. (2024). https://arxiv.org/abs/2312.15223

[26] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS)*. https://arxiv.org/abs/2306.05685