Engineering Reliability into LLM Multi-Agent Systems: An Industrial Experience

Marco Becattini*, Matteo Minin*, Lorenzo Cioni†, Nuno Laranjeiro‡, Roberto Verdecchia*, Enrico Vicario*
*University of Florence, Italy {marco.becattini, matteo.minin, roberto.verdecchia, enrico.vicario}@unifi.it
†Lascaux, Italy lorenzo.cioni@lascaux.it
‡University of Coimbra, Portugal cnl@dei.uc.pt

***Abstract*—A central tension in productionizing LLM agents lies between the intrinsically deterministic requirements of engineering and the stochastic nature of AI models, a tension often aggravated by the unreliability of the data the models consume. In this work, we present an experience on engineering reliability into an LLM Multi-Agent system, namely HorseReport, a production service launched at the 127th edition of the largest equestrian fair in Italy. HorseReport implements five different reliability mechanisms, namely (i) layered acquisition with graceful fallback, (ii) output as a compile-time contract, (iii) bounded, self-repairing compilation, (iv) persistent state with automatic resumption, and (v) granular observability by design. The system relies on a Kafka pipeline to first scrape unreliable sources with progressively more robust fetch strategies, with HTTP first, and browser automation as a last resort, and subsequently run nine LLM agents in two parallel phases, each writing one LaTeX section of the report. Each agent can use only a whitelist of LaTeX commands intended to reduce invalid markup, and the HorseReport pipeline saves its state so interrupted runs resume on their own. As upstream output stabilised, an early LLM self-repair loop was replaced by a deterministic, log-driven fixer that avoids additional model calls. To date, HorseReport has processed 328 reports without requiring any manual intervention, with over 95% of a 100-report sample judged acceptable by domain experts, a median latency of approximately 70 seconds, and an LLM API cost of about €0.18 per report. From the industrial case study, we draw four main lessons learned on engineering reliability into LLM Multi-Agent Systems, namely (i) *constrain LLMs rather than trust them*, (ii) *once a stochastic stage is reliable, prefer determinism*, (iii) *make reliability and cost observable per unit*, and (iv) *unattended operation is a systems property, not a model property*.**

***Index Terms*—Reliability Engineering, LLMs, GenAI, Industrial Experience Report**

I. Introduction

Generative AI promises to automate knowledge work, yet moving large language models (LLMs) from a product demo component into a dependable production service exposes a hard tension, namely, the engineering around LLMs needs predictable behaviour, while the models are stochastic and the external data they consume is often incomplete or ill-structured. For a software-reliability practitioner, an LLM is an unusual component that can produce a different, occasionally invalid, output for the same input, and whose failures rarely raise an explicit error as these often appear as plausible but incorrect output that must still be detected and handled [1]–[3]. The reliability of such nondeterministic components, once integrated with conventional software, has been identified by the software reliability community as a key emerging software engineering challenge [4].

This paper presents an industrial experience report on embedding reliability into software architectures that feature an LLM component. The report considers *HorseReport* as an industrial case study, a production service that autonomously generates multi-page PDF reports on show-jumping horses. Given only a horse identifier, the system gathers data from several external equestrian sources, commissions nine specialised LLM agents to draft the individual sections of a report in LaTeX, compiles the result to a typeset PDF, and delivers the final report, a multi-page dossier covering the horse's pedigree, competition history, a technical analysis and an economic valuation with recommendations, with no human in the loop. HorseReport has been in production since its November 2025 launch at the 127th edition of Fieracavalli, the largest equestrian fair held in Italy, and is publicly available at horsereport.ai.

The central focus of this work is reliability: *Can a pipeline of stochastic agents, fed with unreliable data scraped from the web, run reliably in production without human intervention?* In HorseReport, reliability is engineered around the ensemble of agents rather than what is expected from any single one of them, through a set of recurring reliability mechanisms. In HorseReport, we implement prompt-level constraints intended to reduce invalid LaTeX; bounded automatic correction of residual failures; persistent pipeline state-machine caching allowing interrupted agent runs to resume independently; and per-report monitoring mechanisms to track cost, runtime, and quality.

In production, HorseReport has run unattended, generating hundreds of reports with no manual intervention, at an LLM API cost of roughly €0.18 each. All data presented in this experience report, supported by their analysis and final results, are made available online [5].

The industrial report presented in this work gravitates around three core contributions:

- An in-depth record on engineering reliability into a real, deployed, fully autonomous LLM multi-agent system [6], with operational data from production, including failure and cost figures.
- A catalogue of established reliability mechanisms adapted and combined around an industrial LLM pipeline, each

presented as a reusable problem/solution/trade-off pattern.
- A transferable lesson learned on determinism: As stochastic stages are made reliable, deterministic mechanisms should be preferred over LLM-in-the-loop ones.

II. Industrial Context

In this section, we introduce the application domain that HorseReport addresses and the context in which it operates, laying the groundwork to motivate the design of the system.

A. Domain

A sport-horse report is a structured dossier that buyers, sellers, breeders and agents use to assess a show-jumping horse: its identity and breeding, its competition record and results, statistical trends in its performance, an analysis of its pedigree, and market valuation. Compiling such a report by hand is a slow and tedious specialized task: relevant facts are scattered across several national and international equestrian databases, each with its own format. Turning such data into a clean, consistently typeset document takes hours. HorseReport completely automates this end-to-end data-to-document generation task [7].

The raw information comes from external equestrian sources, principally the Italian federation FISE, the international federation FEI, and the pedigree database HorseTelex, with EquiResult as an additional results source. None of these platforms offers a clean, stable, public API for the data we need. In practice the system retrieves a horse's registry data and competition history by authenticating to and scraping web pages, resolving the animal across databases through an identifier-matching step, and extracting its pedigree both as structured data and as a rendered image. The input data is therefore heterogeneous, paginated, multilingual, and littered with non-standard special characters (ampersands, accents, superscripts) that are innocuous on web pages read by humans, but are prone to breaking downstream typesetters. Cross-database identifier matching is itself error-prone, as two concrete exemplary cases illustrate. A single animal often possesses a fragmented identifier across records: `CARTHUSIA DEL COLLE` (FRA, 2012), for example, appears under three separate FISE profiles (`45593B`, `CERTORIG`, `ORIG`), only one of which carries its FEI identifier `107FY56`, making it hard to uniquely identify the animal. Conversely, a name query such as "furia" returns several distinct horses, some sharing the same birth year, so that the pair (name, birth year) is not a primary key and candidate disambiguation is required.

B. The limits of single-prompt generation

A natural first approach to address our application domain is to treat report generation as a single prompt, namely to hand a model the scraped data and request a document. In our production setting we observed that this strategy fails due to reasons that have little to do with LLM generation abilities. The LLM output is intrinsically non-deterministic and occasionally syntactically ill-formatted. The document we require however must be a valid, compilable artefact, not merely plausible prose [8], [9]. In addition the input data is sometimes partial or inconsistent; and the whole process must run unattended, at predictable cost and latency, recovering from the inevitable transient failures of half a dozen external systems. Our engineering effort has therefore focused on making generation dependable, rather than focusing on the content generation processes alone [10].

III. Our Three Challenges of LLM Reliability Engineering

We organise the rest of the paper around three LLM reliability challenges that we identified as characteristic of our agent-based software-intensive system. We systematically address these challenges in HorseReport *via* different reliability mechanisms, as further documented in Section V.

- **C1: Unreliable external data.** Our input data is scraped from sources rather than served through stable APIs [11]: pages change, sessions expire, requests are rate-limited or time out, and referential integrity constraints may be violated across databases. The system must still obtain consistent data, or fail cleanly, despite flaky inputs.
- **C2: Unpredictable model output.** Each agent must emit not text formatted in natural language but LaTeX content contributing to a shared document. A stochastic model can produce invalid markup, forbidden commands, or unescaped data characters, i.e., defects that are silent until compilation breaks.
- **C3: Long, partially failing processes.** A single report spans scraping, nine agent invocations, assembly and multi-pass compilation, touching several external services. Any stage can fail midway, so the pipeline must track its own progress and resume, rather than restart or stall, while keeping cost and time bounded.

A. Related work.

Reliability is the foremost reported challenge in deploying LLM agents. A recent practitioner study (20 case studies and 86 surveyed practitioners) finds it is addressed not *via* more accurate models but by systems-level design [10]. Accordingly, a recent taxonomy frames LLM failures as a system-engineering rather than a model-centric problem [1]. A complementary line proposes reference architectures and runtime patterns for production agents, including the explicit stochastic–deterministic boundary that motivates our determinism lesson [2], [12]. The narrower problem of forcing valid structured output (underpinning our output-as-contract mechanism) is addressed by constrained and grammar-guided decoding [8], [9] and programmable output guardrails [13]. AIOps work at ISSRE instead applies ML to predict [14] or diagnose [15] failures in conventional systems [16]. In this work, we build upon the existing body of knowledge by sharing a report on our concrete experience on engineering LLM reliability in practice.

IV. System Architecture

This section describes the architecture of HorseReport and how a report is produced, from the acquisition of a horse's data to the delivery of the final PDF.

A. Overview

HorseReport instantiates SALLMA, a reference architecture for LLM-based multi-agent systems [17]. The system is implemented as a tenant of a multi-tenant integration middleware built on Apache Camel, in which each unit of work is an explicit, named route [18]. The system is event-driven: it consumes messages from an Apache Kafka topic, and a message's `op` header selects one of two operations, namely `SCRAPE`, which acquires a horse's data, and `GENERATE`, which turns acquired data into a delivered PDF. Splitting acquisition from generation is a deliberate design choice. The two tasks have different failure profiles and cost structures, and decoupling them lets each be retried, resumed, or replayed independently [19]. Figure 1 shows the end-to-end flow. All configuration (credentials, endpoints, agent identifiers, retry bounds, cost rates) is supplied per tenant from a database parameter map rather than hard-coded, so the same code can be run against test and production environments by configuration.

B. Data acquisition (`SCRAPE`)

On a `SCRAPE` request the system first consults a local cache. If a recent report artefact already exists for a horse, it is recompiled and delivered without contacting any external source. On a cache miss it creates a persistent `Process` record and walks it through a small state machine (`pending` → `fisescrape` → `horsetelexscrape` → `equiresultscrape` → `feiscrape` → `done`), persisting the intermediate data and the current state after each step. Acquisition queries four external sources. The system first authenticates to and scrapes FISE for the horse's registry data and its competition history across the current and previous seasons, handling pagination and session cookies. It then resolves the same horse on HorseTelex and retrieves its pedigree as structured JSON, as parsed text, and as a rendered image. Finally it scrapes two further competition-result sources, EquiResult and FEI, merging their events with those from FISE and de-duplicating across sources with FEI taking precedence. External interactions use progressively more robust fetch strategies: a direct HTTP request first, full browser automation (Selenium) as a last resort, with the exact strategy varying by source, and each wrapped in bounded retries; the detailed failure-handling is reported in Section V. When acquisition completes, the `processId` is attached to the originating customer ticket and the process is marked `done`.

C. Report generation (`GENERATE`)

A `GENERATE` request loads the acquired data by `processId`, verifies tenant ownership, and enriches it with age, statistics, normalised pedigree, genealogy, and image assets. It initialises per-report observability for tokens, cost, elapsed time, and attempts. Nine specialised LLM agents produce a typical report's ten sections: title page; data sheet; pedigree and analysis (one agent); competition history and statistics; technical analysis; valuation; recommendations; and conclusions. Each uses its own prompt and model; production uses *Claude Haiku 4.5* for reliable LaTeX output and convenient pricing. Their authenticated endpoint calls return LaTeX fragments. This decomposition is a design choice, not a necessity claim: it supports section-specific contracts, parallelism, and dependencies; one-shot comparison is future work.

To keep latency low, the agents run in two parallel phases. The first phase consists of six agents whose output is assembled into an initial document; the second consists of three agents (economic valuation, recommendations, conclusions) that depend on that initial content. The assembled LaTeX is written to disk, compiled to PDF, and the finished PDF is uploaded to the customer ticketing system together with the collected cost and quality metrics. The compilation step itself (multi-pass, bounded-retry, self-repairing) is described in Section V.

D. State, resumption and housekeeping

Generating a report relies on several external services over tens of seconds, and any of these steps can fail midway. For this reason the report's state is persisted to a database in the `Process` record, which stores its current step and the data collected so far, so that the report can be resumed rather than restarted after a failure [20]. A deterministic software monitor periodically scans for unfinished reports and resumes each from its last saved step, re-running only missing work, so a temporary external-service failure does not lose the request. A second scheduled route enforces a retention policy on generated artefacts.

V. Reliability Mechanisms

Reliability in HorseReport is not the property of any single component but the cumulative effect of five recurring mechanisms, each addressing one of the challenges of Section II. We present them as a small catalogue of patterns; Table I summarises each as problem, solution, and entailed trade-offs.

M1: Layered acquisition with graceful fallback (C1). *Problem:* External sources are scraped, not served: pages change, sessions expire, requests time out, and a horse key can be ambiguous across databases. *Solution:* External interactions climb a ladder of progressively more robust but more expensive strategies: a direct HTTP request first; the same request replayed with cookies obtained from a real browser session; and, as a last resort, full browser automation deriving the page *via* Selenium. Each acquisition strategy is wrapped in bounded retries with session reuse, cross-database identifier resolution recovers ambiguous horses, and a local cache short-circuits the whole ladder when a recent artefact already exists. *Trade-off:* The fallback strategies are progressively slower and more resource-hungry, and a successful fallback can mask

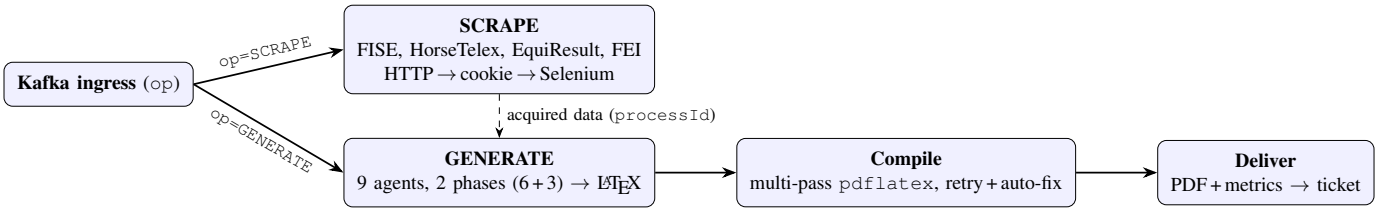

Fig. 1. HorseReport end-to-end flow. A Kafka message's `op` header selects one of two decoupled operations: `SCRAPE` (top) acquires a horse's data, and `GENERATE` (bottom), triggered by a separate message, loads that data by `processId` and runs nine LLM agents in two parallel phases before LaTeX compilation and delivery.

a degrading source, which is precisely why acquisition is instrumented (see M5).

M2: Output as a compile-time contract (C2). *Problem:* A stochastic model asked for a document section can emit invalid markup (unbalanced environments, undefined commands or colours, unescaped data characters) that is silent until it breaks compilation. *Solution:* We do not explicitly ask agents to use LaTeX syntax but rather, we bind them to a contract, a prompt-level mitigation rather than a grammatical guarantee like constrained decoding [8], [9]. Validation has three stages. First, the prompt permits only whitelisted commands, environments, and predefined colours, requires data-character escaping, and forbids `\usepackage`, `\definecolor`, and arbitrary colours. Second, deterministic post-processing removes explanatory text before and after the requested fragment. Third, `pdflatex` compilation is the enforceable check; failure triggers M3. *Trade-off:* The contract narrows stylistic freedom and must be maintained as the template evolves, but it converts an open-ended failure mode into a closed, testable one.

M3: Bounded, self-repairing, compilation (C2). *Problem:* Even under a contract, the occasional malformed fragment slips through, and an uncorrected one fails the whole report. *Solution:* Compilation runs inside two nested bounded loops. An inner loop of up to three compile attempts (each running three `pdflatex` passes to resolve cross-references, with an automatic log-driven repair between attempts), wrapped in an outer loop that, on a compilation failure, re-runs the entire generation (re-invoking all the LLM agents) up to a configurable number of times (three by default). The first production version repaired failures by feeding the compiler error log back to an LLM and asking it to return corrected LaTeX, up to five times, an instance of the LLM self-repair and iterative-refinement paradigm [21], [22]. Once the M2 contract was in place, the agents' output became structured and predictable enough to repair deterministically. We therefore replaced the LLM-in-the-loop repair with a fix that parses the compiler log, locates the offending construct, and closes it: No model call, no extra tokens, no added latency, and a repair that behaves identically every time. The outer loop remains as a bounded backstop: When the deterministic fixer cannot repair a fragment, it re-runs the entire generation, giving the stochastic agents a fresh attempt before the report is declared failed. *Trade-off:* The fixer only recognises encoded failure classes, and successful compilation does not establish semantic preservation. Unknown failures exhaust the bounded attempts and are reported rather than returned as corrupt documents.

M4: Persistent state with automatic resumption (C3). *Problem:* A report spans scraping, nine agent calls, and compilation across several services over tens of seconds; any stage can fail midway. *Solution:* Each report is a persistent process advancing through an explicit state machine, with intermediate results stored after every step. The deterministic software monitor detects processes stuck in a non-terminal state and resumes them from the correct point, re-running only the missing work. A definitive failure instead advances the customer ticket into an explicit failed state rather than leaving it silently pending. *Trade-off:* Correctness depends on each step being safe to re-run and on disciplined state transitions, which adds design effort, but it is what removes the human operator from the loop.

M5: Per-report observability (cross-cutting). *Problem:* Stochastic, externally fed systems can degrade without obvious symptoms. Cost may rise, a data source may become unreliable, and output quality may decline, none of it visible without explicit monitoring. *Solution:* Every report carries an observability record that accumulates token usage, token-derived LLM API cost, elapsed time and retry attempts, and these metrics are delivered alongside the PDF on the customer ticket. Reliability and economics thus become first-class, per-unit data rather than aggregate guesses. *Trade-off:* Observability detects rather than prevents. Its value depends on the metrics actually being watched. It is, however, what made the M3 evolution measurable in the first place.

VI. Evaluation in Production and Lessons Learned

This section presents the results observed during the operation of HorseReport in production and the main lessons drawn from our industrial experience about architecting the system.

A. Deployment

HorseReport has been running in production since November 2025, when it was launched at Fieracavalli, and has since generated 328 reports in routine operation. The evidence we report is therefore operational rather than experimental. We found no like-for-like commercial baseline: public equine services differ in scope, inputs, automation, and human involvement. We assess it the way its operators do (by whether

TABLE I
CATALOGUE OF RELIABILITY MECHANISMS.

Mechanism	Chall.	Problem	Solution	Trade-off
M1 Layered acquisition	C1	flaky scraped sources	HTTP→cookie→Selenium ladder + retries + cache + ID resolution	slower/heavier fallback can mask source decay
M2 Output-as-contract	C2	invalid LLM markup	command/colour whitelist + escaping rules + post-processing	less freedom; contract upkeep
M3 Bounded self-repair	C2	residual compile failures	bounded retry; deterministic log-driven fix (was LLM self-repair)	only encoded failure classes repaired
M4 Persistent state + resume	C3	partial mid-pipeline failure	state machine + software monitor resumes missing steps	steps must be safe to re-run
M5 Observability	all	silent degradation	per-report tokens/cost/time/attempts on the ticket	detects, does not prevent

TABLE II
HORSEREPORT IN PRODUCTION.

Aspect	Observed value
In production since	Nov. 2025 (Fieracavalli)
Reports processed	328
Delivered / failed	316 / 12 (≈96% delivered)
Manual interventions	none
Content quality	>95% (sample of 100)
100-run compilation	81 first-try; 10 fixed; 7 failed; 2 not reached
Request→delivery time	median 70 s, mean 82 s (43–547)
LLM API cost	≈ €0.18/report (token-based; $n = 108$)

it runs without intervention, whether its output is good enough to ship, and whether it is economical), and we are explicit (Section VII) about the limits of the reported evidence.

B. Unattended operation (C1, C3)

Of 328 production requests, 316 were delivered and 12 ended in an explicit failed state, i.e., a 96.3% success rate. No request required manual intervention to be unblocked, corrected, or re-run. Transient failures of external sources do occur, but the layered acquisition (M1) and resumable state machine (M4) absorbed them. A stalled process is resumed from its last valid step, and requests that cannot be completed are advanced into an explicit fail state rather than left silently pending, so failures surface cleanly without human-in-the-loop. Unattended operation is thus the observed mode over the entirety of reports to date, rather than an aspiration.

C. Output quality (C2)

The output contract (M2) and bounded self-repair (M3) make compilation failures rare. Agents generally emit syntactically valid markup, and occasional mechanical defects are fixed deterministically. Three domain experts with over 10 years' experience assessed 100 randomly sampled reports from the 316 production deliveries. Source data were ground truth for factual sections and the proprietary algorithm for valuation; analysis and recommendations lacked a unique reference and were checked for consistency. A report was acceptable only if no objectively verifiable error was found in any section. Discussion yielded consensus; over 95% were acceptable.

D. Cost and latency (C3)

Generation is fully automated and inexpensive. End-to-end from request to delivery, reports take a median 70 s (mean 82 s, range 43–547 s; $n = 316$). This covers scraping, nine agent calls in two parallel phases, compilation, and delivery, rather than a single document-processing step; from token usage, M5 records a mean LLM API cost of about €0.18 ($n = 108$), excluding non-LLM services, infrastructure, and labour. To estimate the fixer's effectiveness, we ran a 100-run batch: 81 compiled first try, 10 were fixed, 7 reached compilation but failed, and 2 failed upstream before compilation. The fixer therefore rescued 10 runs without another model call; we did not compare its repair success, semantic preservation, latency, or total cost against the earlier LLM-based method on the same failures.

VII. THREATS TO VALIDITY

Our evidence is the operational record of one production system. We did not ablate M1–M5, use a controlled baseline, or stress-test throughput; the observations show that the integrated system works in practice but do not isolate individual causal effects, establish optimality, or support a scalability claim. The quality figure rests on a random sample of 100 delivered reports and consensus among three internal and external assessors rather than independent ratings, so it remains indicative rather than a guarantee over the full population. Horse-specific sources, identity resolution, and valuation are domain-specific. The reliability patterns may transfer to other data-to-document systems, but this has not been demonstrated outside equestrian reporting.

VIII. LESSONS LEARNED ON ENGINEERING RELIABILITY INTO LLM MULTI-AGENT SYSTEMS

We distil four lessons that may be transferable beyond our industrial experience report.

Constrain LLMs rather than trust them. When an LLM must produce a structured artefact, reliability comes more from restricting what it is allowed to emit than from adopting a more capable model: turning an open-ended failure mode into a closed, testable one is the most effective approach. In our case, almost all output reliability (C2) came from this narrowing (M2), not from a higher-end model.

Once a stochastic stage is reliable, prefer determinism. Once a stochastic stage is constrained enough that its failures are rare and mechanical, a deterministic handler avoids additional model calls and behaves repeatably; reserve model intelligence for where variability is needed. In HorseReport, the deterministic fixer replaced LLM-in-the-loop repair once the contract made failures predictable, although the two were not directly compared.

Make reliability and cost observable per unit. Per-unit metrics on reliability and cost are what reveal when a stochastic stage has stabilised enough to be hardened. Our per-report records (M5) were what showed that residual failures had become mechanical and the deterministic fixer safe to adopt.

Unattended operation is a systems property, not a model property. Running with no human in the loop comes from classic reliability engineering, namely graceful degradation against unreliable inputs and resumable persistent state, wrapped around the generative core rather than from the core itself [23]. In HorseReport, these were layered acquisition (M1) and resumable state (M4).

IX. Conclusion

In this experience report we described HorseReport, a production service that turns a pipeline of stochastic LLM agents and unreliable web data into a fully autonomous, dependable report generator that has been in operation since November 2025. Its reliability comes not from the models but from the engineering around them: A compile-time output contract, bounded and increasingly deterministic self-repair, layered acquisition with fallback, resumable persistent state, and per-report observability. Our experience yields a lesson we believe might be generalised: *Constrain stochastic components until their failures are rare and mechanical, and then prefer deterministic mechanisms over keeping a model in the loop, reserving generative AI for the work that genuinely needs it*. This structure may transfer to other constrained-syntax generation tasks, but we have not evaluated it outside equestrian reporting.

Our future work spans three directions, the first two drawn directly from the determinism lesson and aimed at handling failures earlier and more locally. First, we plan to push the deterministic repair down to the level of the individual agent, validating and fixing each LaTeX fragment as it is produced rather than only at final assembly, so that a defect is contained to the agent that caused it. Second, we plan to give each agent a compilation tool it can invoke directly, moving error handling into the agent's own loop; localising failure handling in this way would avoid re-running the whole pipeline and further reduce token consumption. Third, we plan a controlled ablation of deterministic versus LLM-based repair, comparing repair success, latency, and cost, and will apply the same patterns to other data-to-document domains.

Acknowledgments

This research paper was developed with the assistance of LLMs to enhance clarity. All text was manually revised.

References

[1] V. Vinay, "Failure modes in LLM systems: A system-level taxonomy for reliable AI applications," *arXiv preprint arXiv:2511.19933*, 2025.

[2] B. Xia, Q. Lu, L. Zhu, Z. Xing, D. Zhao, and H. Zhang, "Evaluation-driven development and operations of LLM agents: A process model and reference architecture," *arXiv preprint arXiv:2411.13768*, 2024.

[3] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, 2023.

[4] P. Pelliccione and N. Laranjeiro, "Insights from the software reliability research community," *IEEE Reliability Magazine*, pp. 10–14, 2024.

[5] M. Becattini, M. Minin, L. Cioni, N. Laranjeiro, R. Verdecchia, and E. Vicario, "HorseReport: Production data and associated results," 2026. [Online]. Available: https://doi.org/10.5281/zenodo.20937326

[6] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," in *Proc. 33rd Int. Joint Conf. on Artificial Intelligence (IJCAI-24)*, 2024, pp. 8048–8057. [Online]. Available: https://www.ijcai.org/proceedings/2024/890

[7] A. Gatt and E. Krahmer, "Survey of the state of the art in natural language generation: Core tasks, applications and evaluation," *Journal of Artificial Intelligence Research*, vol. 61, pp. 65–170, 2018.

[8] S. Geng, M. Josifoski, M. Peyrard, and R. West, "Grammar-constrained decoding for structured NLP tasks without finetuning," in *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

[9] B. T. Willard and R. Louf, "Efficient guided generation for large language models," *arXiv preprint arXiv:2307.09702*, 2023.

[10] M. Z. Pan *et al.*, "Measuring agents in production," *arXiv preprint arXiv:2512.04123*, 2025.

[11] E. Ferrara, P. D. Meo, G. Fiumara, and R. Baumgartner, "Web data extraction, applications and techniques: A survey," *Knowledge-Based Systems*, vol. 70, pp. 301–323, 2014.

[12] V. Srinivasan, "A methodology for selecting and composing runtime architecture patterns for production LLM agents," *arXiv preprint arXiv:2605.20173*, 2026.

[13] T. Rebedea, R. Dinu, M. Sreedhar, C. Parisien, and J. Cohen, "NeMo guardrails: A toolkit for controllable and safe LLM applications with programmable rails," in *Proc. EMNLP (System Demonstrations)*, 2023.

[14] Y. Liu, M. Ma *et al.*, "Early bird: Ensuring reliability of cloud systems through early failure prediction," in *Proc. IEEE Int. Symp. Software Reliability Engineering (ISSRE), Industry Track*, 2024.

[15] E. Yu *et al.*, "HRCA: A heterogeneous graph-based adaptive root cause analysis framework," in *Proc. IEEE Int. Symp. Software Reliability Engineering (ISSRE), Industry Track*, 2023.

[16] P. Notaro, J. Cardoso, and M. Gerndt, "A survey of AIOps methods for failure management," *ACM Transactions on Intelligent Systems and Technology*, vol. 12, no. 6, 2021.

[17] M. Becattini, R. Verdecchia, and E. Vicario, "SALLMA: A prototypical software architecture for LLM-based multi-agent systems," in *Proc. IEEE/ACM Int. Workshop on New Trends in Software Architecture (SATrends)*, 2025.

[18] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[19] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[20] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[21] A. Madaan *et al.*, "Self-refine: Iterative refinement with self-feedback," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. [Online]. Available: https://arxiv.org/abs/2303.17651

[22] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," in *Int. Conf. on Learning Representations (ICLR)*, 2024. [Online]. Available: https://arxiv.org/abs/2304.05128

[23] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.