Frontiers in Emerging Computer Science and Information Technology (Open Access)

Volume 02, Issue 10, October 2025,
Publish Date: 10-21-2025
DOI: - https://doi.org/10.64917/fecsit/Volume02Issue10-04 PageNo.38-45

Building a Low-Code Conversational Platform: Learnings and Architecture Considerations

Mayank Vats
Principal Software Development Engineer, Workday, Pleasanton, CA, USA

RECEIVED - 09-22-2025, **RECEIVED REVISED VERSION** - 09-29-2025, **ACCEPTED**- 10-01-2025, **PUBLISHED**- 10-21-2025

Abstract
Conversational tools are now expected in most enterprise systems, but building them in a consistent and scalable way is still difficult. In many cases, teams end up repeating business logic, managing fragmented NLP models, or struggling with long development cycles. To address these issues, we designed and deployed a Low-Code platform that allowed both developers and nondevelopers to create conversational skills within a single governed framework. A key design choice was to separate natural language processing, orchestration, and business logic, which let the NLP group improve core models while application teams focused only on skill behaviour. The platform grew steadily in use: within three years it handled roughly 300,000 conversations each day; by year five it supported 3,000 tenants; and by year six it was available in nine languages and across six different channels. Development time dropped from months to weeks, while governance, testing, and monitoring were built directly into the framework. This paper shares the architecture, the practical challenges we encountered, and the lessons learned. More broadly, it shows how Low-Code methods can be adapted to enterprise conversational AI, and what this means for future large-scale SaaS and agent-based platforms.

Keywords: *Low-Code platforms, Conversational AI, Natural Language Processing, Orchestration, SaaS ecosystems, Enterprise governance, Multi-tenancy, Observability, Enterprise AI, Scalability*

1. Introduction

Low-Code development has become a popular way for organizations to accelerate software delivery and expand who can participate in the building of applications [7, 6, 11].The appeal is clear: drag-and-drop tools and reusable components make it possible for developers to work faster and for designers and product managers which are nontechnical to prototype ideas without writing much code. In parallel, conversational systems, such as chat bots and digital assistants, are now a common feature of enterprise software, helping employees' complete tasks or find information.

Despite this progress, existing Low-Code platforms and conversational frameworks leave important gaps. General-purpose Low-Code tools like Mendix, OutSystems, or Microsoft PowerApps do not offer the kind of multi-tenant governance, language support, or integration depth needed in large SaaS environments. On the other hand, conversational AI frameworks such as Rasa [4] or Dialogflow provide strong NLP capabilities, but lack enterprise-grade features to scale across thousands of customers, enforce security, or prevent 'skill sprawl'. In short,

today's tools focus on either simplifying development or language understanding, but very few bring the two together in a way that meets enterprise requirements.

This paper describes the design and operation of a platform built to close that gap. The system allowed both developers, designers and product managers to build conversational skills in a unified Low-Code environment, while enforcing governance, testing, and monitoring at scale. A central architectural choice was to separate NLP, orchestration, and business logic, so that machine learning specialists could advance language models without needing to understand business workflows, and application developers could focus on logic without worrying about

NLP internals.

The platform's adoption over time illustrates its impact. Within three years, it was handling around 300,000 conversations per day. By its fifth year, it supported more than 3,000 tenants and by year six it was deployed across nine locales and six communication channels. Most importantly, it reduced delivery times from months to weeks, while ensuring consistency across a wide range of use cases.

By sharing architecture, challenges, and lessons learned, this paper makes two contributions. First, it demonstrates how Low-Code principles can be adapted to conversational AI in an enterprise SaaS setting, where multi-tenancy, governance, and reliability are non-negotiable. Second, it highlights design practices—such as clear separation of concerns and built-in observability—that can inform the next generation of Low-Code platforms.

2. Related Work

Low-Code development platforms have been extensively studied in both industry and academia. Gartner [7] and Forrester [6] have tracked their rapid adoption and positioning in the enterprise landscape. Open-source projects such as Rasa [4] and the spaCy ecosystem [10] have provided foundational tools for understanding natural language, while native frameworks of the cloud such as AWS Step Functions [2] have shaped integration patterns for orchestrating distributed workflows. Reliability practices have been formalized in the discipline of Site Reliability En- gineering [3], which influenced the observability and governance choices of this platform. Fi- nally, Jurafsky and Martin's comprehensive text [8] provides a theoretical basis for the NLP approaches underpinning conversational systems.

3. System Architecture

The platform was built as a collection of microservices to keep it modular and easier to evolve. At the center of the framework was the flow runtime service, which acted as the orchestration layer. It executed conversation graphs, stored skill definitions, and coordinated all API inter- actions. Alongside it, a dedicated NLP service handled intent detection and entity recognition. This service was based on open-source models [4, 8], trained on tenant data which has been de- identified, and deployed in multiple language-specific variants to support international users. To provide a consistent user experience across web, mobile, Slack, Teams, and other channels, responses were normalized through channel adapters that generated generic card components. The high-level architecture of these interactions is shown in Figure 1.

Conversation flows were represented as directed acyclic graphs, with node types such as prompts, API calls, conditions, transforms, sub-flows, native code execution, messages, error handlers, and end states. Validation routines ensured that these graphs were well formed, preventing infinite loops, unreachable nodes, or dead ends. To encourage reuse, the platform

supported subflows and prebuilt templates for common task redirects. Data isolation was handled at the database level, where tenant information was separated by strict row and field-level security.

For integration with back-end systems, the platform relied on an API gateway that supported both synchronous and streaming calls. Reliability was reinforced with circuit breakers, retry logic, and timeouts at the platform layer, while idempotency was left to the underlying APIs. To handle slow services, the team introduced short-lived caches and pre-fetching strategies, which helped reduce latency spikes. Authentication varied by context: JWTs or session tokens were used for user calls, while Alpaca certificates authenticated service-to-service requests. Developer access was managed through Active Directory groups, and all changes were audited [2].

Observability was designed from the start. The platform operated with we operated with a

99.99 percent availability target and kept the 99th-percentile latency below one second, consistent with SRE best practices [3]. It produced metrics and traces for every skill, node, and tenant, and exposed them through Grafana dashboards. In addition, conversation analytics tracked both abandonment and task success, providing insight into real user outcomes. Testing was integrated into the life-cycle of skill development: skill definitions were stored as JSON DSLs, allowing deterministic replays across builds. Promotion followed a controlled pipeline through development, performance, staging and production, with feature toggles enabling gradual roll- outs and safe rollback. The component-level details of this architecture are shown in Figure 2, which highlights the flow runtime core, repository service, NLP connectors, and their integration with external providers

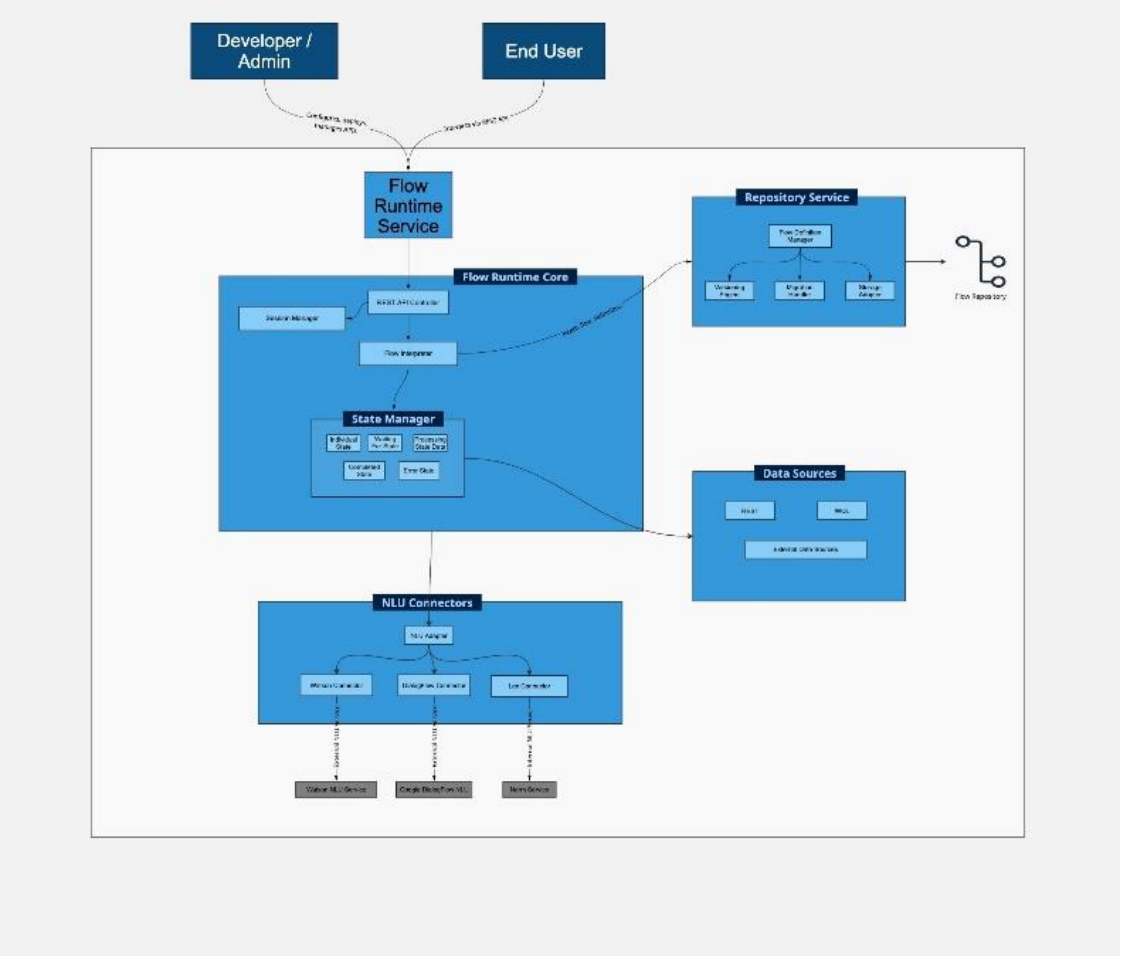

Figure 1: High-level architecture of the Low-Code conversational platform.

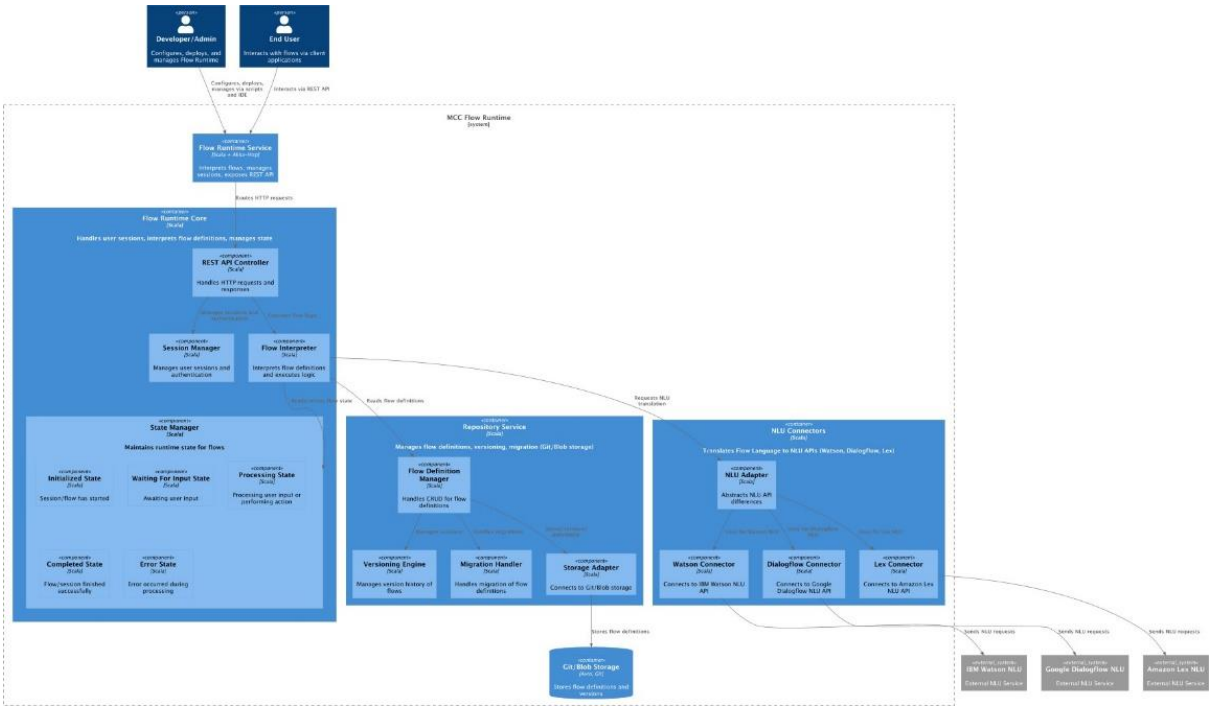

Figure 2: Component-level (C4) architecture showing the flow runtime core, repository service, NLU connectors, and integration with external NLU providers.

4. **Low-Code Authoring and Adoption**

The platform was built with two audiences in mind: professional developers and non-technical contributors such as product managers and quality analysts. To support both groups, it offered two ways to build conversations. The first was a visual, drag-and-drop editor that let users' piece together flows without writing code. The second was a domain-specific language (DSL) that gave developers direct control over the underlying JSON metadata and allowed them to fine-tune complex logic.

In practice, developers gravitated toward the DSL for advanced scenarios, such as tenant- specific logic or multi-turn workflows. Nondevelopers, on the other hand, used the visual editor to create prototypes, proofs of concept, or test flows. To make the editor approachable, it included features like inline validation, typeahead suggestions for entities, and instant feedback. This meant that product managers could draft simple conversational flows themselves, while quality analysts could write automated test cases without depending on engineering teams.

Reusability was another core design principle. Teams could save time by assembling new skills from existing sub flows and reusable API nodes, rather than starting from scratch. The platform also separated conversation logic from localized content, which allowed the same flow to run across multiple languages without duplication. For users who found it difficult to work directly with API responses, the system provided a swagger parser that generated helper functions automatically. These decisions—reusability, separation of logic and content, and built-in guidance—made it possible for both developers and nondevelopers to contribute meaningfully. Over time, this inclusivity helped the platform scale across roles while still meeting enterprise standards of reliability.

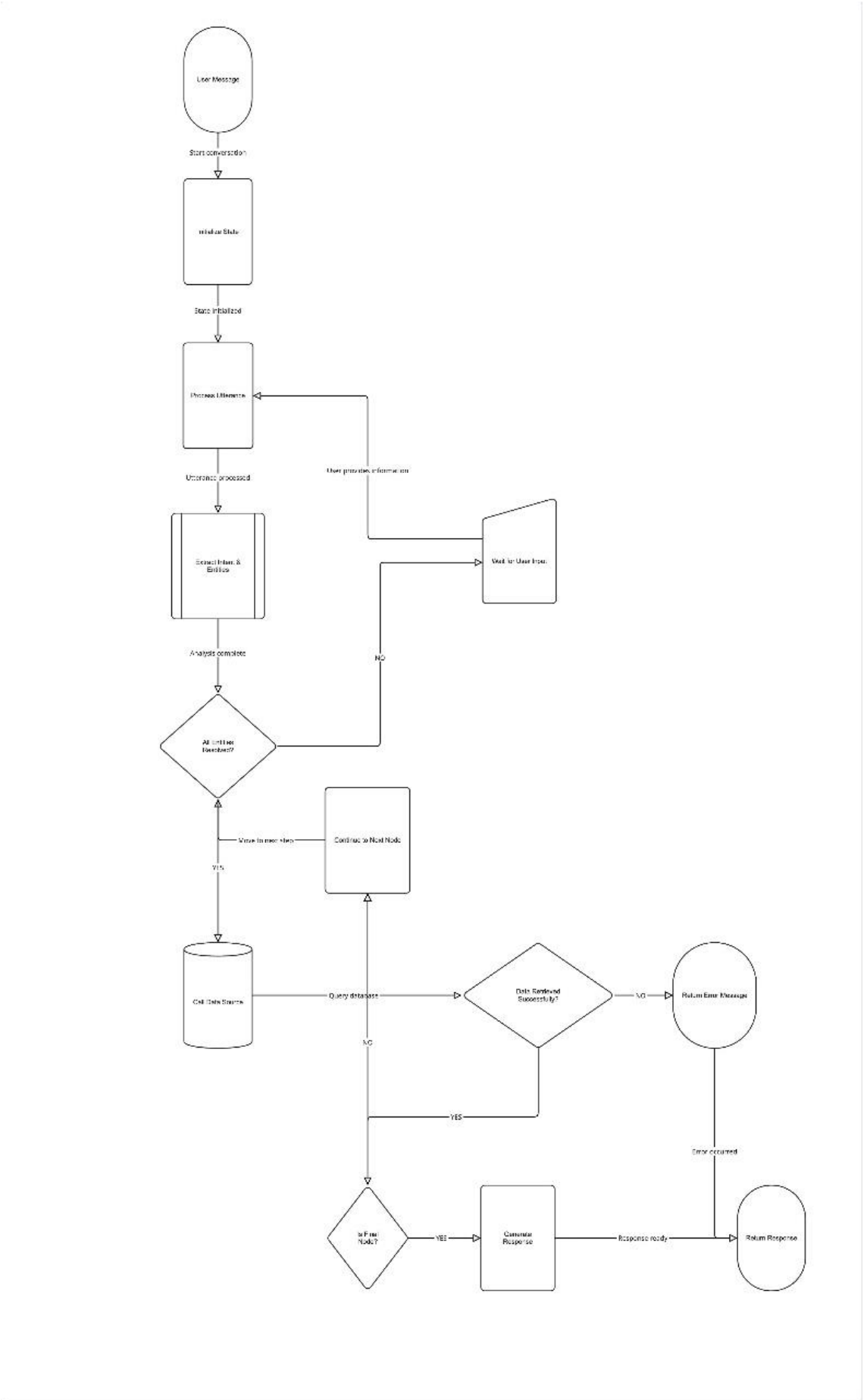

Figure 3: Conversation state flow illustrating initialization, entity extraction, conditional branching for unresolved entities, data retrieval with success and error paths, and final response generation

5. Conversation Flow Runtime

The way a conversation runs on the platform can be thought of as moving through a state machine, illustrated in Figure 3 Every time a user sends a message, a new conversational sequence begins. The system first sets up the conversation state and then processes the user's input. At this stage, the NLP service analyzes the utterance to identify the user's intent and extract any relevant entities.

The platform then checks whether it has everything it needs to continue. If some information is missing, the flow shifts into a "wait for input" state, prompting the user to fill in the gaps. Once all the required entities are available, the conversation advances to the next step in the graph.

Some of these steps involve reaching out to external data sources. If that query fails, the system gracefully handles the error and informs the user. If it succeeds, the conversation continues, moving through intermediate nodes until it eventually reaches a terminal state.

At the end of the flow, the platform composes the final response and delivers it back to the user. This structured approach ensures conversations are predictable and reliable, while still supporting the realities of multi-turn dialogue, error handling, and user interruptions.

6. Challenges and Resolutions

While the platform was originally intended to simplify development, real-world use quickly exposed several challenges. One recurring pain point was the heavy reliance on backend APIs. Some of these calls were slow or resource-intensive, which led to latency spikes and, at times, instability. To handle this, the team introduced a caching layer within the orchestration service so that frequently used responses could be reused for short periods instead of triggering repeated calls. For APIs that were consistently slow, a prefetching mechanism was added to load data in advance, reducing delays and sometimes erroring out from the user's perspective.

Another early difficulty was maintaining context across multiple skills. Without a standard handoff mechanism, the experience could feel inconsistent or even break when moving between flows. This was addressed by creating explicit contracts for context exchange, along with validation checks to ensure that the receiving skill had all the input it needed before execution. Authentication posed its own hurdles, since the platform had to support authentication mechanish as a user and as a service. The team solved this by adopting a standardized approach: JWTs or session tokens for user requests, and Alpaca certificates for service calls, each paired with stricter auditing for compliance.

Inclusivity also brought challenges. Non-developers, particularly product managers and designers, often struggled with tasks like writing expressions or handling raw API responses. This risked undermining the platform's goal of broad participation. To lower the barrier, the system introduced a swagger parser that could automatically generate helper functions, as well as reusable templates that guided users through common patterns. Another obstacle was versioning, there was no easy way to roll back changes or maintain parallel paths in a conversation. This gap was closed using the organization's toggling framework, which allowed multiple paths to exist safely and can be rolled out gradually.

Some of the most important lessons came from production incidents. In one case, a reporting API that had never been optimized for speed was reused for a user-facing feature. The result was that the assistant's landing page began to time out in production. The short-term fix was to quickly disable the feature with a toggle; the long-term solution was to add caching to the reporting API so it could support more demanding use. This experience reinforced the need

for feature flags, phased rollouts, and performance reviews whenever existing services were repurposed.

As adoption grew, governance became essential to avoid chaos. The platform enforced one-to-one mappings between intents and active skills, introduced linting rules, and added integra- tion tests to prevent overlaps or ambiguous flows. These measures curbed the 'skill sprawl' and maintained clarity even as the number of skills scaled. Together, caching, prefetching, standardized contracts, and governance mechanisms stabilized the platform, built user trust, and made it resilient enough to handle enterprise-scale workloads.

7. Results and Evaluation

As adoption grew, the platform began to show clear improvements in scale, reliability, and developer productivity. Within three years, it was handling more than 300,000 conversations every day. By the fifth year, it was in use across 3,000 tenants, and a year later it supported nine different locales and six delivery channels, ranging from web and mobile apps to Slack, Teams, and browser extensions. Despite this growth, the system consistently delivered on its service-level goals - 99.99 percent availability with response times under one second at the 99th percentile. Accuracy also held strong, with NLP models correctly covering more than 80 percent of supported use cases, thanks to deidentified training data and models tailored for each locale.

One of the biggest wins was the boost in developer productivity. Tasks that once took four to five months

could now be built and deployed in about a month. A good example is the Request Time Off flow, which required orchestrating several tenant-specific APIs and handling complex conditions. Using the Low-Code framework, the team was able to deliver it in just weeks. At the other end of the spectrum, a simpler skill like 'User Email Surfacing' - which displayed information alongside related tasks- showed how quickly developers could assemble lightweight flows with conditional logic and access checks. Together, these case studies illustrated the platform's flexibility, from straightforward informational use cases to complex transactional workflows.

Governance improvements were just as important. By enforcing unique mappings between intents and skills, introducing linting rules and CI validations, and using toggles for controlled rollouts, the platform eliminated duplicated logic and reduced the risk of conflicting flows. An- allistics confirmed that these measures had an impact: conversation abandonment rates dropped, task success went up, and end users reported smoother interactions. Taken together, the gains in productivity, reliability, and governance turned the platform into a foundational piece of conversational development within the enterprise.

8. Lessons Learned

Building and scaling the platform taught us a set of lessons that shaped how it evolved. The first and most important was the value of keeping responsibilities clearly separated. By drawing a firm line between NLP, orchestration, and business logic, we avoided the trap of mixing machine learning with application rules. This allowed the NLP team to focus on improving their models, while developers could focus solely on building business-specific flows. That separation proves to be essential for both scalability and long-term flexibility.

Another key lesson was that performance cannot be an afterthought. From the very be- ginning, the system had to include caching, prefetching, circuit breakers, and retries. Without them, there was no way to meet the strict service-level goals we had set. The early incidents drove home the point that we couldn't rely on downstream APIs to always behave: Our plat- form needed to provide its own safeguards.

We also learned how important visibility is for the trust. Clear service-level objectives—99.99 percent availability and sub-second response times—set expectations, but it was the detailed metrics and analytics that made it possible to catch problems before they became failures. Governance was equally important, if not more. Rules like, enforcing one skill per intent, adding lint checks, and validating flows in CI pipelines kept the system from collapsing under its own growth. Feature toggles gave us the flexibility to experiment and roll back safely when needed. [9]

Finally, we realized that making a Low-Code tool inclusive requires more than a drag-and-drop interface. Non-developers needed templates, guardrails, and helper tools to build with confidence. Without them, the promise of democratization would have been hollow. By combining these supports with technical rigor, the platform grew into something both resilient and accessible: a system that could scale with the enterprise while inviting broader participation.

9. Future Directions

Looking ahead, there are several promising ways the platform could evolve, many of which also open new research opportunities. Recent progress in large language models [5, 1] points to the possibility of blending deterministic conversation flows with probabilistic dialog. At the same time, user studies remind us that non-experts often struggle to make full use of these models [12], which raises important questions about how to design supportive tools and guardrails.

A natural step forward is to give customers more control over how conversations are defined. Instead of relying only on presupplied intents, customers could define their own, tailored to their business needs. Extending this further, the platform could provide interfaces for customers to plug in their own intent recognition models and custom skills. This would transform the system from a centrally managed service into a framework for domain-specific conversational ecosystems. Future research here would need to address how customer-driven customization can coexist with platform-wide governance and security.

Analytics is another key area for growth. At present, the system reports task success and abandonment at a broad level. Customers, however, would benefit from detailed, self-service metrics that reveal how their own flows are performing. Such insights would not only help them improve their designs but also provide researchers with data to study which conversational patterns reduce abandonment and increase task completion. An open question is what set of metrics best capture quality from both a developer's and an end user's perspective.

Finally, large language models open the door to new kinds of interactions. Beyond supporting flow authoring and fallback handling, LLMs could be used directly in conversations to enable richer, back-and-forth dialog alongside rule-based flows. This introduces significant research challenges: how to balance deterministic orchestration with model-driven outputs, and how to ensure reliability, safety, and trust in such hybrid systems.

Together, these directions point toward a future where the platform is more than a centralized tool. Instead, it becomes a collaborative ecosystem—one that empowers customers to define and extend their own skills, gives them meaningful analytics to measure success, and leverages advances in LLMs to broaden the scope of conversational experiences.

10. Conclusion

This paper presents the journey of designing and scaling a Low-Code platform for conversational applications. The system's success rests on a few key principles: keeping NLP, orchestration, and business logic separate; engineering for performance through caching, prefetching, and circuit breakers; and embedding observability and governance as first-class features. By adding templates, helper tools, and integrated testing, the platform also opened participation to non- developers, ensuring inclusivity without compromising quality.

The impact was measurable. Within three years, the platform was supporting roughly 300,000 daily conversations. By year five, it was running for 3,000 tenants, and by year six it had expanded to nine locales and six communication channels. Development cycles shortened from months to weeks, while governance mechanisms prevented duplication and skill sprawl. These results established the platform as a reliable model for enterprise-scale Low-Code adoption.

The lessons learned here go beyond this particular case. SaaS ecosystems that aim to build conversational or agentic platforms can draw on the same foundations: modular separation of concerns, performance engineered from the start, rigorous observability, strong governance, and tools that empower both developers and non-developers. Looking ahead, combining these practices with large language models and richer analytics will further expand what Low-Code systems can achieve. The broader message is clear: Sustainable Low-Code platforms are not only a matter of advanced technology, but also of disciplined design and governance.

References

1. Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Floren- cia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *https://arxiv.org/abs/2303.08774*
2. Amazon Web Services. Aws step functions — what is step functions? https://docs. aws.amazon.com/step-functions/latest/dg/welcome.html, 2019.
3. Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.
4. Tom Bocklisch, Joey Faulkner, Nick Pawlowski, and Alan Nichol. Rasa: Open-source language understanding and dialogue management. *https://arxiv.org/abs/1712.05181*
5. Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Lan- guage models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
6. Forrester The forrester wave™: Low-Code platforms for professional developers, q2 2025. https://www.forrester.com/report/the-forrester-wave-tm-Low-Code-platforms-for-professional-developers- RES182327, 2025. Client access required.
7. Gartner. Worldwide Low-Code development market to grow 23% in 2021 —gartner. https://www.information-age.com/worldwide-Low-Code-development-market-grow-23-2021-gartner-17569/, 2021. Press coverage summarizing Gartner forecast figures. Original Gartner report is paywalled.
8. Daniel Jurafsky and James Martin. Speech and language processing (3rd ed. online manuscript). https://web.stanford.edu/~jurafsky/slp3/, 2025.
9. James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. *MartinFowler. com*, 25(14-26):12, 2014.
10. Luis J. Miranda, Á kos K á d á r, Adriane Boyd, Sofie Van Landeghem, Anders Søgaard, and Matthew Honnibal. Multi hash embeddings in spacy. *https://arxiv.org/abs/2212.09255*
11. Daniel S á, Afonso Lobo, Jo ã o Cunha, Ricardo Duarte, Tiago Guimar ã es, and Manuel Fil- ipe Santos. A state-of-the-art of intelligent problem-oriented Low-Code systems. *Procedia Computer Science*, 2025.
12. J Diego Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. Why johnny can't prompt: how non-ai experts try (and fail) to design llm prompts. In *Proceed- ings of the 2023 CHI conference on human factors in computing systems*, pages 1–21, 2023.