Computers, Materials & Continua

Tech Science Press

Doi:10.32604/cmc.2026.075777

ARTICLE

Automating the Initial Development of Intent-Based Task-Oriented Dialog Systems Using Large Language Models: Experiences and Challenges

Ksenia Kharitonova[1], **David Pérez-Fernández**[2], **Zoraida Callejas**[1,3] **and David Griol**[1,3,*]

[1]Department of Software Engineering, University of Granada, Granada, Spain
[2]Department of Mathematics, Universidad Autónoma de Madrid, Madrid, Spain
[3]Research Centre for Information and Communication Technologies (CITIC-UGR), University of Granada, Granada, Spain

*Corresponding Author: David Griol. Email: dgriol@ugr.es

Received: 08 November 2025; Accepted: 30 December 2025; Published: 12 March 2026

ABSTRACT: Building reliable intent-based, task-oriented dialog systems typically requires substantial manual effort: designers must derive intents, entities, responses, and control logic from raw conversational data, then iterate until the assistant behaves consistently. This paper investigates how far large language models (LLMs) can automate this development. In this paper, we use two reference corpora, Let's Go (English, public transport) and MEDIA (French, hotel booking), to prompt four LLM families (GPT-4o, Claude, Gemini, Mistral Small) and generate the core specifications required by the RASA platform. These include intent sets with example utterances, entity definitions with slot mappings, response templates, and basic dialog flows. To structure this process, we introduce a model- and platform-agnostic pipeline with two phases. The first normalizes and validates LLM-generated artifacts, enforcing cross-file consistency and making slot usage explicit. The second uses a lightweight dialog harness that runs scripted tests and incrementally patches failure points until conversations complete reliably. Across eight projects, all models required some targeted repairs before training. After applying our pipeline, all reached ≥70% task completion (many above 84%), while NLU performance ranged from mid-0.6 to 1.0 macro-F1 depending on domain breadth. These results show that, with modest guidance, current LLMs can produce workable end-to-end dialog prototypes directly from raw transcripts. Our main contributions are: (i) a reusable bootstrap method aligned with industry domain-specific languages (DSLs), (ii) a small set of high-impact corrective patterns, and (iii) a simple but effective harness for closed-loop refinement across conversational platforms.

KEYWORDS: Task-oriented dialog systems; large language models (LLMs); RASA; dialog automation; natural language understanding (NLU); slot filling; conversational AI; human-in-the-loop NLP

1 Introduction

Text and spoken dialog systems are key interfaces for human-computer interaction, especially when users wish to accomplish concrete tasks using natural language [1–3]. Task-oriented and intent-based systems underpin applications such as customer support, self-service information portals, and transactional services, where accuracy, state tracking, and reliable access to structured data are critical [2,3]. Historically, these systems have followed supervised, data-centric workflows grounded in annotated corpora and domain-specific specifications. As applications expand in domain coverage, multilinguality, and regulatory constraints, the need for scalable methods to draft, validate, and maintain these specifications has become central. Recent toolchains increasingly incorporate large language models (LLMs) to accelerate authoring, yet developers must still preserve deterministic control and platform compatibility.

Over the last decade, intent-based architectures have dominated production deployments for task-oriented agents [1,2,4]. A shared design pattern appears across platforms: intents capturing user goals; entities/slots storing task variables; and dialog control encoded as DSLs in the form of stories, rules, flows, or graphs. These structures enable predictability, auditability, and safer error handling than free-form, generative agents. However, they also introduce engineering overhead: curating intent taxonomies, collecting and labeling training examples, synchronizing cross-file resources, and preventing contradictions across interdependent control mechanisms. Open-source and commercial frameworks (e.g., RASA, Dialogflow, Amazon Lex, IBM Watson Assistant, and OpenDial) offer homologous abstractions, differing mostly in file formats and tooling. This commonality suggests that tool-aware but platform-agnostic methods can generalize across the ecosystem.

Recent work has explored how LLMs can support dialog-system creation. Approaches include flow and intent extraction directly from raw dialogs or domain documents [5–7]; bootstrapping and distilling task-oriented agents from instructions, demonstration dialogs, or synthetic data expansions [8–10]; LLM-assisted specification repair, such as resolving contradictions or adding missing control elements in hybrid systems [11,12]; and anchored or constrained generation that grounds LLM outputs in tool schemas, validators, or flow graphs [13–15].

These methods show that LLMs can accelerate early-stage development, but they also highlight persistent gaps: LLM outputs often violate platform constraints, produce cross-file inconsistencies, or generate dialog flows that break during execution. Moreover, evaluation is usually based on Natural Language Understanding only (NLU-only) metrics or structural validity—not on whether the resulting agent can complete tasks end-to-end. Thus, tool-aware execution and debugging—critical in production settings—remain underexplored relative to high-level generation.

Despite this rapid progress, there is no systematic, platform-agnostic method that transforms raw dialogs or one-shot LLM generations into executable, cross-file-consistent artifacts that can be validated, trained, and run directly in standard intent-based toolchains. Prior work mostly addresses isolated steps—intent clustering, flow induction, NLU augmentation, or DSL generation—without forming a closed loop that generates tool-compatible artifacts, diagnoses structural defects using the target platform's own validators and trainers, and links these repairs to measurable improvements in multi-turn task completion. This gap is particularly acute for multilingual setups and legacy domains where dialog logs are noisy or incomplete.

This paper asks how far contemporary LLMs can automate the first mile of building intent-based, task-oriented dialog systems in a framework-agnostic way. Concretely, we focus on three challenges: (i) turning raw dialog logs and one-shot LLM generations into structurally sound, tool-compatible artifacts; (ii) repairing systematic defects in those artifacts without ad hoc, project-specific engineering; and (iii) linking file-level edits to measurable improvements in end-to-end conversations rather than only offline NLU metrics.

Our main contributions are:

1. A reusable, tool-aware bootstrapping method that transforms unstructured, turn-level dialogs from two classic corpora—*Let's Go* (EN) and *MEDIA* (FR)—into consistent, trainable artifacts across platforms: intents with examples, entities/slots with explicit mappings, response templates/actions, and minimal, conflict-free dialog control (forms/rules/flows), all emitted from a single instruction that enforces cross-file invariants. This addresses the structural challenge above by ensuring that LLM outputs can be parsed, validated, and executed by standard tooling (e.g., validators, trainers) without manual reconstruction of the domain schema.

2. A compact set of high-yield micro-patches[1] (intent/action closure, form-first slot collection with explicit mappings[2], and atomic rules that eliminate rule-story/flow contradictions) that reliably convert noisy LLM outputs into coherent, runnable projects. These micro-patches directly target the most common failure modes surfaced by the tools (cross-file drift, slot-mapping gaps, policy contradictions), providing a small, reusable catalogue of fixes rather than bespoke engineering per project.
3. A lightweight, platform-agnostic dialog harness with scripted tests that surfaces operational gaps (start/collect/confirm/complete/recovery) and drives measurable gains in completion, median turns, and help-recovery—thus linking file-level generation to conversation-level success. This harness addresses the gap between static specifications and actual behavior by turning typical task flows into scripted tests, so that the impact of structural fixes and micro-patches can be quantified in terms of task completion and interaction quality.

We have evaluated four model families-GPT-4o, Claude Opus, Gemini 2.5 Pro, and an open-source (OS) baseline (Mistral Small)-spanning major proprietary stacks and a deploy-anywhere OSS option that produced usable artifacts. Across eight projects (4 models × 2 corpora), Phase I (validator bootstrap) made all snapshots trainable; Phase II (harness-guided stabilization) brought them to ≥70% task completion (many 84%–100%) with short median turns and near-perfect help recovery. Although we demonstrate concrete outputs in RASA, the method operates over an *intermediate schema* and a swappable renderer, making it portable to other intent-based platforms (e.g., mapping to Dialogflow intents/entities/flows or to Lex intents/slot types and elicitation prompts). Taken together, tool-aware bootstrapping, micro-patches, and the dialog harness form a minimal closed loop: generate artifacts, diagnose and repair systematic issues with the platform's own tools, and verify that these repairs translate into robust task-level behavior.

The remainder of the paper is structured as follows. Section 2 reviews related work on intent-based agents, LLM-assisted development, and hybrid architectures. Section 3 describes the main components of the RASA platform and how to use them to develop conversational systems. Section 4 (Materials and Methods) describes the corpora, our platform-agnostic generation template and provider token budgets, Phase I validator-driven bootstrapping, and Phase II harness-guided stabilization. Section 5 reports results: NLU cross-validation, robustness by linguistic slices, confusion structure, test-based interaction KPIs, LLM-as-a-judge evaluation of final test conversations, and integral human evaluation of final snapshots by two experts in RASA chatbot development. Section 6 discusses why LLMs do not yet fully automate end-to-end bot creation, the minimal interventions that consistently work, and limitations. Finally, Section 7 concludes with implications for closed-loop, tool-aware generation[3] and broader, multilingual deployments across frameworks.

2 Related Work

Task-oriented dialog systems have traditionally been built using intent-based architectures, supported by both commercial platforms and open-source frameworks [1–4]. Platforms like Google Dialogflow, IBM Watson Assistant, and Amazon Lex provide cloud-based toolkits for designing intent classifications and dialog flows, often via graphical interfaces or forms. In parallel, open-source frameworks such as RASA[4] [16]

[1] Small, localized configuration edits (e.g., adding a missing intent to `domain.yml`, tightening a `slot_mapping`, or splitting one conflicting rule into two atomic rules) that fix a specific failure mode without redesigning the overall domain schema or dialog policy.
[2] Once a task intent is detected, control is handed to a single governing form that elicits all required slots in a fixed sequence using explicit `slot_mappings` (e.g., `from_text`/`from_entity`) while the slot is requested, instead of distributing slot questions across multiple, competing rules or stories.
[3] LLM-based code and specification generation that is explicitly conditioned on the target platform's tools and constraints, so that the model produces artifacts designed to be checked, run, and iteratively repaired by those tools rather than free-form text in isolation.
[4] https://rasa.com/.

and OpenDial[5] [17] offer developers greater control and transparency. RASA in particular has become a popular toolkit for building conversational agents with modular components for NLU (intents/entities) and dialog management (stories or rules). It requires developers to manually define a domain (intents, slots, actions) and provide example training phrases and conversational stories. This intent-driven paradigm ensures deterministic behavior and easier error handling, but it demands substantial up-front effort in data annotation and flow design. As described in [18], choosing between various development tools often involves trade-offs between ease-of-use and flexibility: low-code platforms streamline development at the cost of transparency, whereas code-centric frameworks like RASA offer extensibility but carry a steeper learning curve.

Early approaches to dialog system construction were largely rule-based or template-driven, requiring developers to anticipate all possible user inputs and dialog paths. The advent of statistical and machine learning methods enabled data-driven dialog models, including end-to-end neural systems that learn from large dialog corpora. Notably, the release of multi-domain conversational datasets such as MultiWOZ [19] demonstrated that fully data-driven task-oriented dialog models can be trained to handle complex dialogs across domains. However, purely end-to-end models demand extensive annotated data and tend to struggle with the accuracy and consistency required in real-world applications. In practice, industry systems often still rely on explicit intent and slot definitions (the pipeline architecture) to maintain reliability. Developing these intent taxonomies and dialog rules is labor-intensive: domain experts must manually craft numerous example utterances for each intent and enumerate conversation branches for slot-filling and error handling. This manual design process can lead to rigid interactions that are brittle when users stray from anticipated inputs. Recent studies underscore the difficulty of covering the vast variability of natural language; even well-structured intent-based bots can feel artificial or overly sensitive to phrasing variations [20].

To alleviate the design bottleneck, researchers have explored methods to automatically extract or generate dialog system specifications from unstructured data. One line of work is intent induction or intent mining from conversation logs. Instead of starting with a predefined intent list, intent mining algorithms attempt to discover latent intents by clustering similar user utterances from historical dialogs. For example, a pipeline that analyzes past chat transcripts to group semantically related utterances is proposed in [21], thereby suggesting candidate intent categories and training examples. Such data-driven intent discovery can bootstrap an initial NLU model, reducing the reliance on manual brainstorming of intents. Another approach focuses on automatic dialog flow generation. A dialog generator for a smart home assistant is described in [22] to automatically generate a RASA story graph (dialog flows) from high-level specifications. Their system uses a tree expansion algorithm to enumerate possible conversation paths (user intents and system responses) and generates the RASA`stories.yml` training file accordingly. This method illustrates the potential to algorithmically derive structured conversation rules from relatively abstract domain knowledge. Similarly, some commercial design tools attempt to ingest example dialogs or FAQs and output draft conversational flows. However, fully automating the design of a domain-specific dialog agent remains challenging: it is difficult for algorithms to anticipate all nuances of human interaction, and auto-generated content often requires substantial manual refinement by conversation designers.

The emergence of LLMs has opened a new avenue for automating dialog system development. LLMs like GPT-3 [23], GPT-4 [24], Anthropic Claude [25], and others have demonstrated an unprecedented ability to understand and generate natural language, including the capacity to perform zero-shot or few-shot task adaptation. This has led researchers to explore using LLMs both as components within dialog systems and as tools to assist developers. One direct application is using an LLM as the dialog manager or response

[5] https://github.com/plison/opendial.

generator in place of a hand-crafted policy. For instance, a chatbot entirely through prompting GPT-3 with carefully engineered instructions (a system dubbed "BotDesigner") is described in [26]. In this approach, the developer writes prompt templates that guide the LLM on how to handle user inputs, effectively outsourcing the dialog policy to the LLM's generative capabilities. This no-code method lowers the effort to produce natural, varied system responses. Nonetheless, follow-up work [27] highlights significant challenges: non-expert designers often struggle to craft effective prompts, and even with good prompts, the LLM's behavior can be unpredictable or difficult to constrain.

Recent large-scale evidence further sharpens this concern: the study completed in [28] shows that across 200,000+ simulated conversations and 15 models, performance in multi-turn, underspecified settings drops by about 25 absolute points (from ~90% single-turn to ~65% multi-turn), an average relative degradation of ~39% across six generation tasks. They attribute the gap less to raw aptitude and more to unreliability: models make premature assumptions, propose final solutions too early, and then over-commit to earlier (possibly incorrect) states, failing to recover. Thus, while LLMs inject flexibility and human-like fluency, relying on them in a free-form manner risks losing the deterministic control that traditional frameworks provide-motivating hybrid, tool-aware approaches (like ours) that preserve schema discipline, cross-file consistency, and explicit control over slot-filling and confirmation.

A recent trend is to combine the strengths of LLMs with the structured rigor of intent-based systems. One example is the approach by [29], which integrates an LLM into the dialog pipeline to interpret user utterances into a domain-specific semantic representation. In their system, the LLM essentially acts as a parser: it translates raw user input into a logical form or dialog act (expressed in a predefined DSL) aligned with the developer's business logic. This preserves a clear separation between understanding and action execution—the LLM handles NLU flexibly, while deterministic rules or code manage dialog state over the LLM-produced frames. Complementary work pushes structure into memory and tooling: *MemGuide* performs intent-driven memory selection to stabilize multi-session, goal-oriented agents [30], while *GenieWorksheets* offers a program-by-worksheet methodology for building reliable, testable task+knowledge agents with explicit control over tools and state [31]. In parallel, semi-supervised joint SLU (domain, intent, and slot modeling) has been shown to improve data efficiency and robustness by leveraging unlabeled chat logs [32]. Together, these hybrid and semi-supervised strategies reduce manual engineering compared to conventional intent-based bots while mitigating the unpredictability of unconstrained LLM agents.

In parallel, tool builders and researchers are investigating LLM-assisted development: using LLMs to generate the artifacts needed by frameworks like RASA. The goal is to automatically produce initial versions of files such as `nlu.yml` (intent definitions with example utterances), `domain.yml` (slots, entities, and responses), and `stories.yml` (dialog flow stories) from unstructured dialog data or simple specifications. Our work follows this direction, and related efforts are beginning to appear in the literature. A model-driven chatbot development approach extended with GPT-3 has been recently proposed in [33] for generating more natural responses. By integrating an autoregressive LLM into a model-driven engineering workflow (in their case, the Xatkit DSL platform), they show that even a partially "scripted" bot can benefit from the generative power of LLMs to handle open-ended user inputs or to polish the assistant's phrasing. The combination of a formal dialog model with an LLM aims to yield a system that is both robust and expressive.

Likewise, a method for declaratively creating task-oriented chatbots using LLMs is proposed in [20], effectively blending intent-based and LLM-based paradigms. They argue that traditional intent-based chatbots often result in rigid, artificial dialogs because developers cannot feasibly anticipate every phrasing or user behavior, whereas LLMs can produce more fluid and contextually adaptive interactions. However, LLM-only bots are typically general-purpose and lack built-in domain boundaries. To narrow an LLM's behavior to a specific task domain, one can employ fine-tuning on domain data or retrieval augmentation techniques,

but these introduce additional complexity and cost [20]. Complementary data-centric approaches further reduce manual authoring: SynTOD generates synthetic task-oriented dialogs from explicit state-transition graphs, enabling controllable coverage of intents, slots, and multi-turn flows [34]; in a related vein, self-talk bootstrapping has LLMs role-play both user and agent to produce multi-turn training conversations that can be filtered and used for supervised fine-tuning [9]. Together, these LLM-assisted artifact and data-generation strategies provide practical pathways to seed `nlu.yml`, `domain.yml`, and `stories.yml` with minimal human effort while keeping a structured, intent-based backbone.

As described in this section, the current state-of-the-art in dialog system development is a spectrum: on one end, fully rule-based or intent-based systems offer reliability but require extensive manual design; on the other end, LLM-driven agents promise rapid development and rich language usage but pose challenges in control and accuracy. The research community is actively exploring middle-ground approaches that use LLMs to automate the tedious aspects of dialog design (like generating training examples or handling linguistic variations) while keeping a coherent structure that ensures the chatbot stays on track. This related work survey shows that our effort-automating the creation of RASA bot files from unstructured dialogs via LLMs-aligns with emerging trends. We build on prior findings that LLMs can dramatically speed up the bootstrapping of conversational agents, yet we also heed the documented limitations of LLM-based methods (e.g., potential hallucinations, inconsistency, and the need for careful prompt engineering). Our work contributes to this line of research by providing a systematic evaluation of LLM-generated dialog components (intents, stories, domain) on real corpora (Let's Go, MEDIA) and highlighting where human intervention is still required to reach production-quality performance.

3 The RASA Conversational AI Platform

As previously described, task-oriented dialog systems are traditionally grounded in intent-based architectures, which have been widely adopted in industrial and research applications due to their predictable behavior, transparent state tracking, and compatibility with structured backend services [3,35–37]. Within this paradigm, RASA emerged as a prominent open-source framework that integrates both rule-based and machine-learned components for NLU and dialog management [16,38]. Rather than introducing novel primitives, RASA operationalizes long-established ideas from the NLU and dialog management literature, providing a modular environment suitable for reproducible experimentation and large-scale deployment.

From a scientific perspective, RASA can be understood in terms of the functional subsystems shown in Fig. 1, which map closely to established models of conversational AI:

1. NLU subsystem: performs intent recognition and entity extraction, consistent with semantic frame-based approaches in NLU [39,40].
2. Dialog management subsystem: maintains dialog state and selects system actions using a combination of rules and learned policies, aligning with statistical dialog management research [37,41,42].
3. Action subsystem: executes domain-specific operations through predefined or custom actions, enabling integration with external systems.
4. Tracker subsystem: maintains a structured sequence of events representing conversational context, reflecting principles from dialog state tracking literature [43].

Developing and deploying a chatbot in RASA involves a structured workflow that integrates data preparation, model training, testing, and deployment. Each phase ensures that the system can understand natural language inputs (RASA NLU), manage conversations intelligently (RASA CORE), and respond appropriately through well-defined actions:

- Data preparation corresponds to defining semantic frames and communicative intents.

- Pipeline configuration enables comparative studies across feature extractors, tokenizers, and classifiers.
- Policy configuration allows the evaluation of rule-based, memory-based, or neural policies under consistent conditions.
- Evaluation tools provide metrics for intent accuracy, entity F1, and dialog coherence, consistent with Dialogue State Tracking Challenge (DSTC) evaluation methodologies [43].

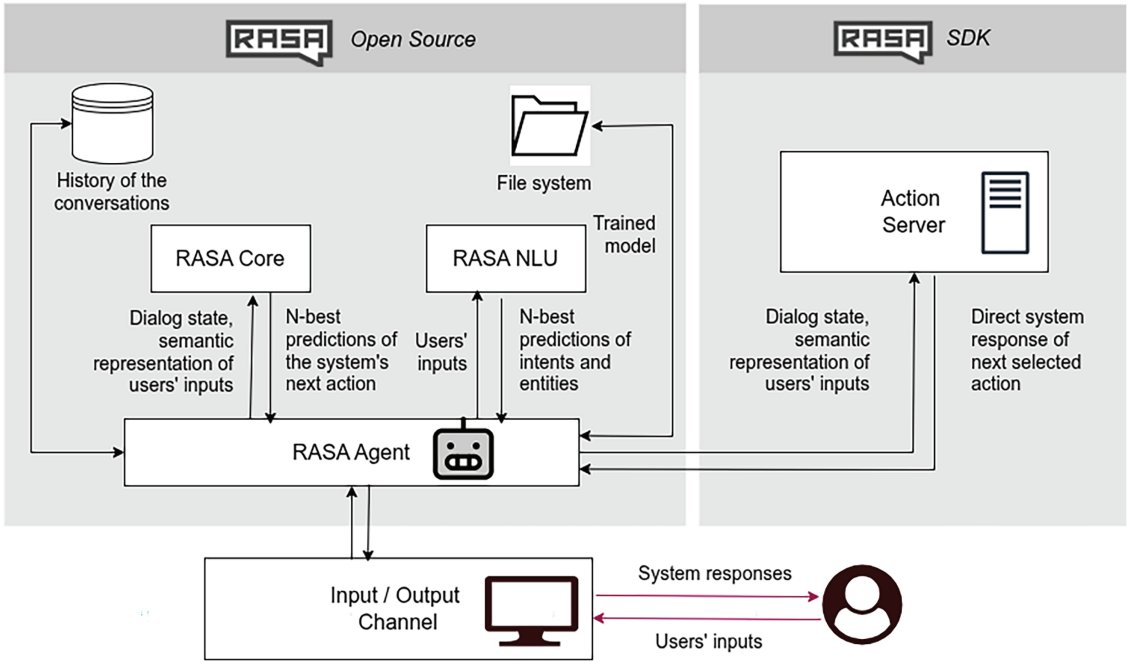

Figure 1: Architecture of RASA

The first step in building a RASA chatbot is defining the domain, intents, entities, slots, responses, and actions. This configuration provides the foundational structure of the assistant. The domain definition is specified in `domain.yml` and describes all elements the bot can handle (intents, entities, slots, responses, actions forms, and session configuration). NLU training data is stored in `data/nlu.yml`, containing example user utterances labeled with intents and entities. Dialog training data is defined in `data/stories.yml` and `data/rules.yml`, representing the expected conversation paths.

RASA uses a processing pipeline (configured in `config.yml`) that defines how user input is analyzed. The pipeline is a sequence of components that progressively transform text into structured meaning (intents and entities). Each component has a specific purpose: tokenizer (breaks text into tokens), featurizer (converts text into numerical representations), classifier (predicts intents and entities), mapper (handles synonyms and normalization), and response selector (helps with FAQ-style responses). The combination of these components can be customized depending on the language, domain, and complexity of the chatbot.

Early NLU research conceptualized user understanding as a joint problem of intent classification and slot filling [44], a formulation that has influenced commercial virtual assistants and academic benchmarks alike [39,40]. RASA's NLU subsystem follows this tradition through configurable pipelines composed of tokenizers, featurizers, and classifiers, which can be combined or extended to evaluate different modeling strategies. Intent classification and entity extraction in RASA are supervised learning tasks grounded in prior NLU work on semantic parsing and slot filling [45]. Slots populated during NLU feed into the dialog

state, enabling multi-turn goal resolution. Dialog policies in RASA CORE compute next actions by conditioning on this state representation. Rule-based policies offer determinism suitable for safety-critical tasks; memory-based policies replicate previously learned conversation paths; neural policies such as Transformer Embedding Dialog (TED) policy generalize across diverse contexts using transformer embeddings [38].

Dialog management has similarly evolved from finite-state machines and handcrafted rules [35] to statistical and neural approaches capable of generalizing beyond seen conversation paths [42]. RASA CORE embodies this hybrid paradigm by providing deterministic `RulePolicy`, memory-based `MemorizationPolicy`, and neural policies such as the TED [38]. This mixture allows controlled behavior where required, while enabling generalization in multi-turn dialogs. Forms and custom actions represent task-completion mechanisms aligned with slot-filling paradigms studied since early systems such as Automatic Terminal Information Service (ATIS). Although reinforcement learning is not part of the default training loop, RASA's extensibility allows integration with RL-based dialog optimization methods explored in prior literature. The policy configuration is defined in the `config.yml` file.

To train RASA CORE, developers provide stories and rules (i.e., structured examples of how conversations should unfold). Together, stories and rules provide the training data from which RASA CORE learns the logic of the conversation. Stories (defined in `data/stories.yml`) are sequences of intents and actions that represent real or expected dialogs between the user and the bot. Rules (in `data/rules.yml`) define strict, predictable behaviors. Rules are typically used for standard responses that should always behave the same way, independent of conversational context.

Actions are the units of behavior that RASA CORE can execute. They represent what the bot does in response to the user. There are several types of actions. Utter actions (`utter_`) are predefined messages written in the `domain.yml` file that send text responses to the user. Custom actions are Python functions implemented in `actions.py` that can perform logic such as API calls, database queries, or computations. Finally, form actions are specialized actions that manage slot filling by asking for required user information step by step.

RASA's connectors, event brokers, and action servers facilitate integration with production environments, enabling research on dialog behavior in realistic usage conditions. Its transparent data structures and local processing capabilities are particularly relevant for studies requiring fine-grained control, such as analyses of robustness, fairness, adversarial behavior, and human-in-the-loop correction [46]. Fig. 1 provides a summary of the described workflow and set of files required to develop a RASA-based conversational chatbot.

4 Materials and Methods

In this paper, we present a repeatable, domain-independent, and model-agnostic procedure that turns raw dialog logs and LLM-generated YAML into operative RASA 3.x assistants. Our method proceeds in two phases: first, a structural bootstrap that normalizes files, enforces domain-stories-NLU closure, adds explicit slot mappings, and replaces brittle multi-step rules with minimal determinism (atomic rules or a single form) until projects validate and train cleanly; second, a harness-guided stabilization that drives scripted tests to expose interaction failures and applies small, declarative fixes until each bot reaches a passably working state (completion $\geq 50\%$, typically $\geq 70\%$). To keep comparisons fair, we standardize a compact, dependency-free `config.yml` with Dual Intent and Entity Transformer (DIET) on word/character n-grams, export one dialog per row with token counts, avoid external services during bootstrap, and preserve the LLM-proposed intent/slot domain schema: the complete set of intents, entities/slots (with explicit slot mappings) and canonical response/action names proposed by the LLM. Edits are strictly structural, prioritizing file hygiene and conflict resolution over model-specific tuning.

Fig. 2 summarizes the two-phase pipeline: the structural bootstrap first produces a clean, consistent, trainable assistant configuration, and the subsequent harness-guided phase then tests, diagnoses, and incrementally fixes interaction failures until the system behaves reliably in operation.

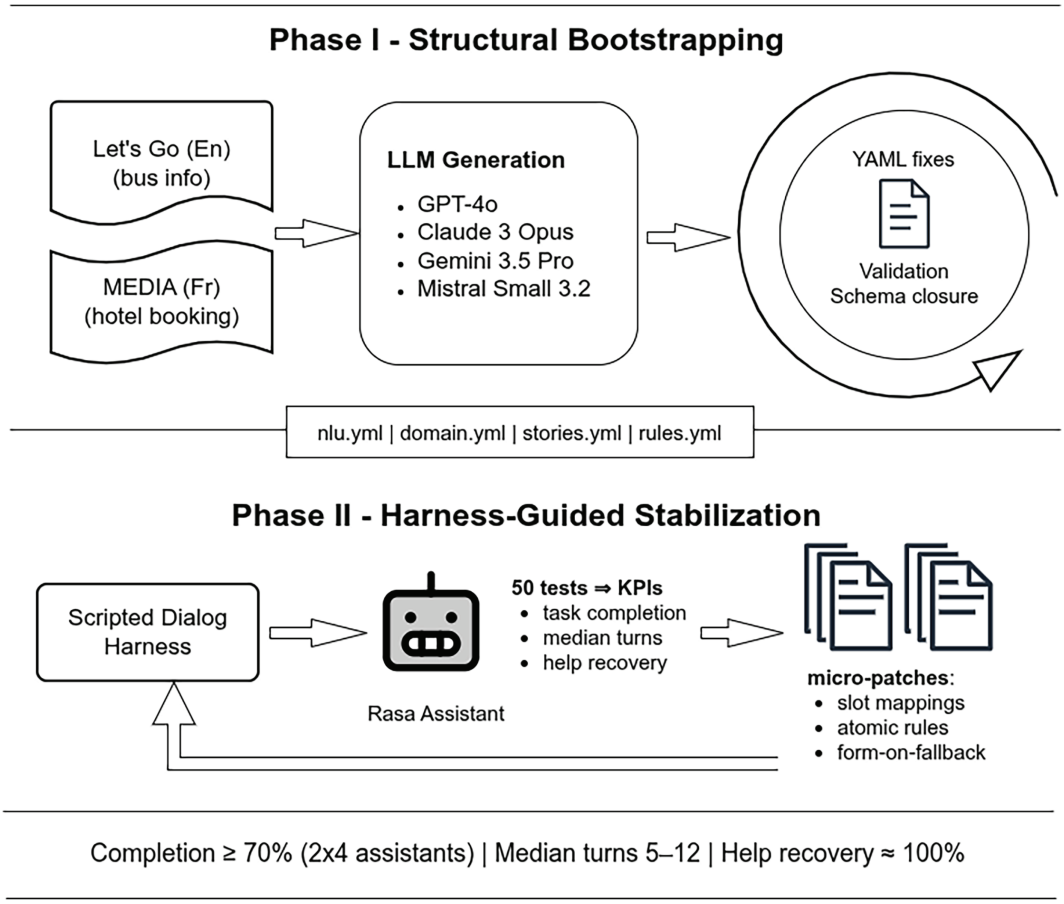

Figure 2: Two-phase pipeline for bootstrapping task-oriented dialog assistants with LLMs

4.1 Dialog Corpora

We have evaluated our bootstrapping pipeline using two reference task-oriented dialog corpora that differ in language, ontology, and interaction style. The English *Let's Go* corpus targets public transportation assistance (bus routes, timetables, trip planning), while the French *MEDIA* corpus covers tourist information and hotel booking. Using the same data preparation philosophy for both-produce clean, linearized dialogs that preserve turn order and realism (Automated Speech Recognition or ASR artifacts, hesitations)—allows us to compare how LLM-generated RASA projects behave across domains and languages.

4.1.1 Let's Go (Public Transportation Assistance)

The *Let's Go* corpus [47,48], originally developed at Carnegie Mellon University, contains real-world telephone interactions with an automated bus information system deployed to the public. Users ask for next buses, previous buses, routes (e.g., `61A`, `28X`), and departure/arrival times. The data is rich in realistic speech phenomena: partial words, self-corrections, barge-ins, re-prompts, ASR errors, and short fragmentary turns

(e.g.,“downtown”, “6:30 p.m.”). We have used the LEGO v2 release (University of Bamberg), provided as CSV, to build LLM prompts and RASA artifacts. Dialog exchanges focus primarily on:

- Trip constraints: origin, destination, and departure/arrival time;
- Route selection: bus line IDs (`28X`, `61A`, `54C`), confirmation/repair;
- Follow-up queries: “next bus”/“previous bus”, new queries, help, and goodbye.

Fig. 3 shows a dialog example extracted from the *Let's Go*. The main challenges specific to this corpus are:

- Short fragments (≤2 tokens) increase confusion among `greet/help/affirm/deny/inform`.
- Route codes behave like entities with narrow lexical support and benefit from lookups/regex.
- Temporal phrases appear in diverse surface forms (“6:30 p.m.”, “tomorrow morning”), where a normalizer (e.g., Duckling) or explicit paraphrases can help; we purposely avoid external services in the bootstrap phase.

Welcome to the CMU Let's Go bus information system.
To get help at any time, just say Help or press zero.
What can I do for you?
SCHEDULE
For example, you can say, when is the next 28X from DOWNTOWN to THE AIRPORT? or I'd like to go from MCKEESPORT to HOMESTEAD tomorrow at 10 a.m..
WHEN IS THE NEXT 28X FROM DOWNTOWN TO THE AIRPORT
Leaving from DOWNTOWN . Did I get that right?
YES
Right
Going to THE AIRPORT . Is this correct?
YES
Right
I think you want the next bus . Am I right?
YES
Alright
The 28X . Did I get that right?
YES
Right
Hold on. I'll look that up.
I'm sorry, but there is no 28X between DOWNTOWN and THE AIRPORT for the rest of the day.
To ask about a different trip, you can say, start a new query. If you are finished, you can say goodbye.
START A NEW QUERY
...

Figure 3: *Let's Go* dialog example (ID 2061122002)

4.1.2 MEDIA (French Tourist Information & Hotel Booking)

We have also used the *MEDIA* speech database (ELRA-S0272), a French corpus of telephone dialogs for tourist information and hotel reservation, distributed by ELDA/ELRA.[6] The collection contains ~1258 calls with `.WAV` audio and human `.TRS` (Transcriber XML) transcriptions. The conversations in the corpus cover:

- Destination and neighborhood constraints (city/arrondissement, proximity to landmarks);
- Temporal and duration constraints (arrival/departure dates, number of nights);
- Room configuration (single/double, rooms, adults/children);
- Amenities/policies (pool, jacuzzi, pets, parking, price);
- Offer presentation and confirmation (listing hotels, quoting prices, availability, booking).

[6] https://catalogue.elra.info/en-us/repository/browse/ELRA-S0272/.

Compared to *Let's Go*, *MEDIA* uses a richer slot space and more varied temporal expressions, and it is in French (diacritics, rich morphology), which stresses multilingual NLU. Fig. 4 reproduces a typical *MEDIA* dialog extracted from the corpus.

bienvenue sur le serveur MEDIA système d'informations touristiques et de réservation d'hôtel souhaitez-vous écouter le guide
oui
ce serveur vocal vous permet d'obtenir des informations touristiques sur les principales villes de France et d'effectuer des réservations d'hôtels ce service fonctionne avec de la reconnaissance automatique de la parole parlez naturellement quelle information voulez-vous obtenir
réservation d'hôtel
où souhaitez-vous aller
Nancy
à Nancy pouvez-vous me donner vos dates d'arrivée et de départ
vingt-quatre mars trente mars
à Nancy du vingt-quatre mars au trente mars dans quel quartier désirez-vous séjourner
près du musée et d'un sauna
veuillez patienter nous recherchons votre information à Nancy du vingt-quatre au trente mars trois hôtels correspondent à votre demande l'hôtel américain l'hôtel Ibis Nancy Sainte-Catherine l'hôtel Mercure Nancy voulez-vous des détails ou réserver dans l'un de ces hôtels suivants
...

Welcome to the MEDIA voice server for tourist information and hotel reservations. Would you like to hear the guide?
Yes.
This IVR lets you get tourist info on major French cities and make hotel bookings. It uses automatic speech recognition, please speak naturally. What information would you like?
Hotel reservation.
Where would you like to go?
Nancy.
In Nancy, can you give me your arrival and departure dates?
March 24th to March 30th.
In Nancy, from March 24 to March 30. Which neighborhood would you like to stay in? **Near the museum and a sauna.**
Please hold, searching your information. In Nancy, 24–30 March, three hotels match: Hôtel Américain, Ibis Nancy Sainte-Catherine, Mercure Nancy. Would you like details or to book one of these?
...

Figure 4: *MEDIA* example dialog (French) and its English translation

We intentionally standardize the export format so that the same prompt templates, LLM providers, and RASA-generation scripts can be reused with minimal localization (French language flag in `config.yml`; amenity lexicons). This enables apples-to-apples comparisons across language and task. The main challenges specific to the *MEDIA* corpus are:

- French locale & diacritics: reliable encoding repair and normalization are essential for clean NLU training examples.
- Temporal variation: multiple date/period paraphrases (*du 24 au 30 mars*, *la quatrième semaine de septembre*) challenge small data; we avoid external normalizers during bootstrap and rely on character n-grams + paraphrases.
- Richer ontology: more slots and intents increase the risk of cross-file drift in LLM-generated YAML; validator-driven closure is crucial.

4.2 Automated RASA File Generation from Unstructured Dialogs

In our proposal, we convert raw, turn-level dialogs into three RASA artifacts (`nlu.yml`, `domain.yml`, and `stories.yml`) using a single structured instruction that encodes cross-file invariants. The methodological goals are: (i) coverage (extract intents, entities, slots, responses, actions, and story flows directly from dialogs), (ii) consistency (exact agreement of intent/entity inventories across `nlu.yml` and `domain.yml`, explicit slot types/mappings, conflict-free stories), and (iii) deployability (artifacts that validate/train in RASA, enact plausible flows, and reflect the task evidenced by the source data). For *MEDIA* specifically,

we *translated the prompt to French* and adapted examples and label names to the tourist/hotel domain (amenities, dates/periods, neighborhoods), while keeping the same file layout and invariants used for *Let's Go*. These cross-file constraints are motivated by the growing evidence that large language models frequently produce structured outputs that deviate from target schemas (e.g., missing fields, malformed JSON/YAML, or inconsistent types) when asked to generate multi-part specifications, which necessitates explicit validation and repair layers around tool-using LLMs [49].

4.2.1 Prompt Design (Model-Agnostic, Language-Adaptable)

We use one instruction template that explicitly asks the LLM to generate the three files and to obey cross-file constraints. The template is language-agnostic: for *Let's Go* (EN) we use the English version (Fig. 5); for *MEDIA* (FR) we supply an automatic French translation and substitute domain-appropriate examples (e.g., *piscine*, *jacuzzi*, *arrivée/départ*, *quartier*). The prompt requires: (1) intents with representative examples; (2) entities and consistent associations to intents; (3) slots with types and explicit `slot_mappings`; (4) response templates and action names; and (5) logically consistent story paths that avoid rule/story contradictions. The same schema is applied to all providers to enable like-for-like comparisons. No few-shot YAML exemplars are prepended in any of the generation scripts; we rely instead on the fact that sufficiently capable LLMs can synthesize the required RASA artifacts from the natural-language specification (RASA being a widely documented, open framework).

```
I have dialogs from a spoken dialog system involving interactions between the system and users. I need to extract three Rasa
files from these dialogs: nlu.yml, domain.yml, and stories.yml. The goal is to ensure there are no inconsistencies in the files,
especially avoiding story structure conflicts and inconsistent intents.

nlu.yml:

1. Extract intents from the dialogs and list example utterances for each intent.
2. Identify entities mentioned in the dialogs and provide examples of entity values.
3. Ensure each entity is consistently associated with the correct intents.

domain.yml:

1. Define intents, entities, slots, and their mappings based on the dialogs.
2. For each slot, include its type and any relevant mappings (e.g., from entities or custom logic).
3. List responses that the system should use, based on the dialogs.
4. Include any actions mentioned in the dialogs.

stories.yml:

1. Extract story structures from the dialogs, ensuring that each story is consistent and follows a logical flow.
2. Avoid conflicts in story structures and ensure that the transitions between intents are clear and coherent.

Please process the following dialogs and generate the required Rasa files:

Dialogs:
{dial}

Requirements:

1. Ensure that the intents in nlu.yml match those in domain.yml.
2. Make sure the slots in domain.yml have appropriate mappings and types.
3. Validate that the story structures in stories.yml do not conflict and logically follow from the dialogs.
```

Figure 5: Prompt template for file generation in English

By pushing all models through the same schema and invariants, we can attribute downstream differences in behavior to known limitations of LLM-based structured generation—such as schema drift, partial outputs, or local inconsistencies across files [49]—rather than to idiosyncratic prompt engineering. This uniform interface is especially important in the multilingual setting, where prior work on multilingual Semantic Language Understanding (SLU) and Named Entity Recognition (NER) reports persistent performance gaps and resource asymmetries between English and other languages, including French [50,51].

In practice, the French adaptation for *MEDIA* deliberately keeps the instruction logic and file invariants identical to the English version while localizing only the surface forms: we translate the meta-instructions and label descriptions into French, inject French-domain lexical items (e.g., *piscine*, *taxe de séjour*, *du 3 au 5*), and keep the same requirements on intents, entities, slots, and stories. This design choice was intentional: it minimizes language-specific prompt engineering and makes the *MEDIA* results primarily a test of each model's multilingual abilities (under a fixed schema) rather than of bespoke French prompt tuning. Our robustness-slice analysis in Section 5.4 reflects this: while the overall pipeline behaves similarly across *Let's Go* and *MEDIA*, models tend to show lower macro-F1 and more brittle behavior on French date-ish and short-token slices, in line with prior observations that multilingual NLU remains harder outside English [50,51].

More broadly, we view this language-adaptable template as a pattern for other domains and languages: the core instruction—"generate consistent NLU, domain, and stories under explicit invariants"—remains unchanged, while the language-, culture-, and domain-specific pieces live in the examples and label glosses. For high-resource languages supported well by current LLMs, we expect our approach to transfer with minimal changes, as long as the prompt is translated carefully and domain lexicons are localized. For morphologically rich, script-divergent, or low-resource languages, additional constraints may be needed (e.g., language-specific tokenization in RASA, stronger guidance on slot boundaries, or light synthetic augmentation), but the overall two-phase pattern-LLM generation → structural bootstrapping → harness-guided repair-remains applicable. Likewise, although we instantiate the artifacts as RASA files, the same prompt pattern can target other platforms (e.g., Dialogflow CX, Lex, Bot Framework) by swapping the requested output schema while reusing the multilingual, instruction-driven extraction paradigm.

4.2.2 LLMs Evaluated and Selection Rationale

We have evaluated four contemporary LLM families that together span the dominant proprietary offerings and a strong open-source baseline. Together, they cover the most widely used proprietary stacks (OpenAI, Anthropic, Google) and a credible open-source option (Mistral) that produced structurally valid YAML. This diversity lets us (i) test generation under different effective context budgets, (ii) compare cross-file consistency and policy coherence across vendor styles, and (iii) evaluate a deploy-anywhere OSS path that many teams prefer for privacy, cost, or integration reasons.

OpenAI GPT-4o. A multimodal successor in the GPT-4 line emphasizing fast, high-quality text generation and long context handling. We chose GPT-4o as a widely used, general-purpose baseline with mature tooling and large effective context for batch generation, making it a natural reference point for cross-provider comparisons [52].

Anthropic Claude 3 Opus. Anthropic's flagship Claude 3 model focused on reliability, instruction-following, and safety with competitive long-context capabilities. We selected Opus to cover the Claude family's strengths in structured outputs and cautious decoding, which are pertinent to cross-file YAML consistency under constraints like rate-limited effective context [53].

Google Gemini 2.5 Pro. Google's long-context, multimodal series are optimized for tool use and grounded text synthesis. We included Gemini 2.5 Pro for its stable 1*m* context window in practice and

strong performance on fragmented inputs, enabling like-for-like runs against our token budgets and French prompts [54].

Mistral Small 3.2 (open source). A compact, instruction-tuned model from Mistral AI deployable behind an OpenAI-compatible endpoint or `llama.cpp`, with efficient inference and permissive licensing.[7] We purposefully added an open-source model to assess whether a lean, locally hostable system can yield acceptable RASA artifacts; Mistral Small 3.2 met this bar and served as a practical OSS baseline [55].

In terms of computational resources, all open-source runs used an on-premise university server from the TrustBoost project to host the Mistral Small model, while proprietary models were accessed via commercial APIs.

With regard to sensitivity to the choice of LLM, our additional experiments indicate that the pipeline requires models that are both sufficiently capable and stable under long prompts to produce fully validatable artifacts at all. Gemini 1.5 Pro, for example, repeatedly looped on `nlu.yml` and never completed a full bundle, whereas Gemini 2.5 Pro removed this failure mode and yielded viable triplets for most chunks, and Claude and GPT-4o did so for all chunks under the same template. On the open-source side, several candidates (e.g., Qwen 3 30B) failed to generate a valid set of files in one shot. Overall, below a certain capability and robustness threshold the pipeline breaks already at artifact generation, but once this bar is met (GPT-4o, Claude, Gemini 2.5 Pro, Mistral Small 3.2), Phase I/Phase II behavior is relatively consistent across LLM families.

With regard to provider limits, we have processed dialogs in batches whose total token count remains below a conservative budget. Given context C, response reserve S, and fixed prompt overhead P, the per-batch budget is $B = C - S - P$. We accumulate dialogs until adding the next would exceed B; the batch is then sealed. Oversized single dialogs ($>B$) are flagged for separate handling (no truncation). The budgets actually used in this study (dialog tokens including prompt $P = 800$ and response $S = 2000$) are:

- GPT-4o: practical dialog window ~84k (context ≈128k, reduced by prompt and provider-side counting quirks).
- Claude Opus: hard context ≈30k; in practice we pace to 10–12 k per batch-equivalent due to rate-limit gating and TPM (tokens-per-minute) caps.
- Gemini 1.5/2.5 Pro: practically usable with 128 k dialogs per batch without issues.
- Mistral Small 3.2: default 4096 output tokens and an input character guard; effective batch comparable to ~30k context.

For each batch, we have spliced the dialogs into the template (EN for *Let's Go*, FR for *MEDIA*) and invoke the model at low stochasticity (temperature ≈ 0.2) to encourage determinism and cross-file alignment. We store the exact prompt and raw response for auditability, then parse the response into:

- `nlu.yml` (intents with examples, entity mentions/values),
- `domain.yml` (intents, entities, explicit slots with `slot_mappings`, responses, actions),
- `stories.yml` (coherent paths consistent with the dialog evidence).

Apart from the language of examples and labels (FR vs. EN), we reuse the same generation files and invariants across both corpora.

Independently of provider and language, every batch must: (i) compile as valid YAML; (ii) close the domain schema triangle (identical intents/entities across NLU and Domain); (iii) define explicit `slot_mappings` (no implicit autofill); and (iv) avoid rule/story contradictions by keeping stories minimal and deferring determinism to forms or atomic rules during bootstrap. These constraints are directly

[7] https://huggingface.co/mistralai/Mistral-Small-3.2-24B-Instruct-2506.

aligned with RASA's recommendations for production assistants, which emphasize explicit slot mappings[8], unambiguous form behavior[9], and avoiding conflicting rules and stories[10] as preconditions for stable policies. These guarantees minimize manual follow-up and ensure each project is immediately ready for structural validation, training, and harness-guided stabilization.

4.3 From Raw LLM Artifacts to Trainable RASA Bots (Phase I: Structural Bootstrapping)

This section describes our first-phase bootstrapping procedure for converting noisy LLM-generated YAML into trainable, coherent RASA 3.x projects. It merges lessons from the initial *Let's Go*–only study (four projects: GPT-4o, Claude Opus, Gemini 1.5/2.5, Mistral Small) with the expanded eight-project, two-corpus effort spanning both *Let's Go* (en) and *MEDIA* (fr). The goal in Phase I is purely structural: make each LLM snapshot validate, train, and behave sensibly in `rasa shell nlu`. Working with both an English and a French corpus also allows us to test our pipeline under multilingual SLU conditions, where prior work has documented that morphologically richer and less-resourced languages tend to exhibit lower intent and slot performance than English, even when using strong pre-trained encoders [50,51].

To keep comparisons fair across models and corpora, we avoided renaming intents, introducing heavy pipelines, or adding external services during Phase I. Edits were strictly structural: YAML hygiene; domain-stories-NLU closure; deterministic control via forms or atomic rules; explicit slot mappings; small hygiene toggles (DIET similarity constraint; span tidying). Applied consistently, this pipeline converted eight heterogeneous, imperfect LLM outputs (four models × two corpora) into trainable RASA projects that performed well under `rasa shell nlu`. This emphasis on explicit slot mappings and deterministic control structures mirrors RASA's own guidance on forms and slots, which discourages implicit autofill and recommends modeling slot collection as transparent, rule-governed flows to avoid unpredictable behavior in production.

In this phase, "automation" refers mainly to systematic diagnosis and re-checking: all edits were made manually based on RASA validator and training logs, while `rasa data validate -debug` and `rasa train -debug` were scripted and rerun until non-benign errors disappeared. We did not implement automatic code rewriting or learned repair; rather, we designed patch types and the validation loop so that each fix is a small, deterministic transformation that could, in principle, be scripted in future work (e.g., adding missing `utter_*` templates or canonicalizing intent names based on validator output). This stable foundation made Phase II (harness-driven interaction loops to raise task completion) straightforward to apply across projects, without per-model ad hoc engineering. The main design principles designed to complete this phase are:

1. Structure over cleverness. Normalize layout, fix YAML, and close the domain-stories-NLU triangle before touching modeling choices or data augmentation.
2. Determinism early. Use atomic one-turn rules (greet/help/bye) and/or a single form for multi-slot flows to eliminate policy contradictions.
3. Minimal edits. Preserve the LLM's domain schema and phrasing; remove external services (e.g., Duckling) unless actually running.
4. Validator loop. Iterate with `rasa data validate -debug` until warnings are benign (unused items, span nits).

[8] https://legacy-docs-oss.rasa.com/docs/rasa/domain/#slot-mappings.
[9] https://legacy-docs-oss.rasa.com/docs/rasa/forms.
[10] https://legacy-docs-oss.rasa.com/docs/rasa/writing-stories/.

We stop once a project: (i) validates with no structural errors, (ii) trains end-to-end (warnings limited to benign items, e.g., span alignment), (iii) parses core tests in `rasa shell nlu`: high-confidence `greet`, sensible mapping of short slot fragments (e.g., "to oakland", "from downtown", "6:30 p.m.") to the intended `provide/inform` labels, and robust resolution of full task requests (e.g., `request_bus_schedule`, `request_hotel_search`). Table 1 summarizes the main failures detected in this phase and the solutions adopted to fix them.

Table 1: Cross-cutting failure modes and micro-patches

Failure	Symptom	Patch
Cross-file drift (names/coverage)	Intents/actions appear in `stories.yml`/`rules.yml` but are missing in `domain.yml` or `data/nlu.yml`.	Canonicalize names (e.g., `request_schedule` vs. `request_bus_schedule`); enforce strict intent/action closure across NLU, domain, and stories.
Rule–story contradictions	`InvalidRule` errors from divergent next-actions prescribed by rules vs. stories.	Prefer a *single form* for multi-slot flows, or reduce rules to *atomic* one-turn behaviors to remove policy conflicts. RASA's rule policy and conflict-checking utilities surface these problems explicitly (e.g., as `InvalidRule` or story conflict warnings), and recommend simplifying or restructuring rules and/or stories to eliminate contradictions.
Missing/unused `utter_*`	Stories reference templates absent from the domain; domain accumulates unused responses.	Add missing `utter_*` templates; wire referenced responses or prune unused ones to keep the domain tidy.
RASA 3.x slot semantics	Silent slot-filling failures due to deprecated implicit autofill assumptions.	Declare explicit `slot_mappings` (use `from_text` for bootstrap; tighten later with `from_entity` under form conditions). This follows the documented transition in RASA 3.x from implicit entity-to-slot mappings to explicit `slot_mappings`, which are required for reliable slot behavior in forms and rules.
External extractors not available	Duckling-related warnings/timeouts at runtime.	Remove Duckling (Phase I) and re-enable only when a live server is provisioned during the quality pass.
Token/annotation misalignment	DIET span warnings (e.g., "7:30 a.m." includes punctuation inside the entity).	Move punctuation outside entity spans; normalize frequent patterns to align spans with tokenizer boundaries.

Table 2 provides a coarse quantification of the error types and manual repairs encountered in Phase I. All eight projects required at least one cross-file drift fix (P1), confirming that name/coverage inconsistencies are a universal by-product of tool-unaware LLM generation. Rule–story contradictions (P2) affected 5/8 projects, in particular *MEDIA* GPT-4o/Claude, *Let's Go* GPT-4o/Claude, and *MEDIA* Mistral Small, reflecting the difficulty of generating non-conflicting policies across rules and stories. Missing or unused `utter_*` templates (P3) appeared in 5/8 projects, and were especially common in the more verbose *Let's Go* snapshots

and in *MEDIA* Gemini/Mistral. Slot-mapping issues (P4) surfaced explicitly only in the GPT-4o projects on *MEDIA* and *Let's Go* (2/8), whereas external extractor problems (P5) and token/span misalignment (P6) were confined to the *Let's Go* Mistral Small project, where Duckling references and span formatting required cleanup. The "Schema validates" column shows that projects typically needed between 1 and 9 validation–repair cycles (median ≈ 6), with the highest counts for *MEDIA* Mistral Small and *MEDIA* Claude Opus, and similarly elevated effort for *Let's Go* GPT-4o.

Table 2: Phase I patch coverage across projects. P1–P6 correspond to the micro-patches in Table 1: P1 = cross-file drift, P2 = rule–story contradictions, P3 = missing/unused `utter_*`, P4 = explicit slot mappings, P5 = external extractors, P6 = token/annotation misalignment. ✓ indicates that the patch type was applied at least once for that project during Phase I. "Schema validates" counts the number of times `rasa data validate -debug` and `rasa train -debug` surfaced non-benign issues that led to fixes in Phase I

Corpus	Model	P1	P2	P3	P4	P5	P6	Schema validates
MEDIA	GPT-4o	✓	✓	–	–	–	–	2
	Claude Opus	✓	✓	–	✓	–	–	8
	Gemini 2.5 Pro	✓	–	✓	–	–	–	1
	Mistral Small	✓	✓	✓	–	–	–	9
Let's Go	GPT-4o	✓	✓	✓	✓	–	–	8
	Claude Opus	✓	✓	✓	–	–	–	6
	Gemini 2.5 Pro	✓	–	–	–	–	–	2
	Mistral Small	✓	–	✓	–	✓	✓	6

Because Phase I was driven by repeated runs of `rasa data validate -debug` and `rasa train -debug`, individual line-level edits were not logged; however, the patch coverage and validation counts together indicate that most snapshots were brought to a trainable state with a small number of patch types and a modest number of validator-guided iterations, without introducing new dialog flows or domain schemas. In practice, each repair consisted of localized YAML changes (e.g., aligning a handful of intent names, simplifying a rule, or adding a missing `utter_*`), chosen manually in response to concrete validator messages. For example, a typical `InvalidRule` message indicating a missing `utter_ask_slot` can be resolved by adding the corresponding response block in `domain.yml`; a cross-file drift warning pointing to an intent present in `stories.yml` but absent from `data/nlu.yml` is resolved by either adding a minimal NLU example or pruning the unused story; a Duckling-related warning is resolved by removing the extractor configuration from `config.yml`. This quantification addresses the level of manual intervention needed for structural bootstrapping and highlights that the bulk of automation in Phase I lies in systematic diagnosis via RASA tools rather than automatic code rewriting.

4.4 Harness-Guided Bootstrapping to Working Prototypes (Phase II)

In Phase II we move beyond structural sanity (Phase I) to *operational* robustness: each LLM-initialized project was iteratively exercised with a unified test harness (`measure_kpis.py`), logs were analyzed, and targeted patches were applied until the agent completed scripted dialogs reliably. The scripted tests were designed as canonical task templates for each corpus, that is, covering the main goal types, required slot combinations, and edge cases such as mid-conversation `help` requests or early goodbyes, so that a single harness could be reused across models. In other domains, the same pattern can be set up by defining domain-specific goals and slot preconditions while keeping the test structure unchanged.

Applying this harness → patch → re-test loop brought all projects to ≥70% completion (several exceeded this substantially), median turns within ~10–18, and help recovery ≥50%. Together, Phases I–II constitute a reproducible path from *raw LLM YAML* to *passably working*, measurable prototypes.

The harness starts each RASA project on a free port, performs a readiness test, emits 50 scripted conversations ("tests")[11], and summarizes conversation-level KPIs. It logs each conversation with final trackers and computes: (i) Completion rate (% of tests where required slots/actions appear in the final tracker), (ii) Median turns, and (iii) Help recovery (completion among "help"-detour tests). We distilled a practical, model-agnostic loop that repeatedly turned sub-performing agents into working prototypes:

1. Run the harness and freeze evidence. Launch `measure_kpis.py`, capture the per-conversation JSON and console summary (completion, median turns, help recovery).
2. Classify the failure from logs. Inspect the last bot utterances, the active loop, and the `requested_slot`. Typical symptoms:
   - *Silent start* or empty warm-up replies,
   - *Form ask → fallback → loop* on short user tokens,
   - *Rule/Story tug-of-war* after form submit,
   - *Wrong slot roles* (e.g., "from X to Y" parsed backwards),
   - *Custom action dependency* stalling the harness (no action server).
3. Apply a minimal, local patch. Prefer declarative changes over code:
   - Add `from_text` mappings for every form slot while that slot is requested,
   - Reduce rules to atomic one-turn edges; let a single form govern multi-slot flows,
   - Replace custom actions with utter-only stubs on submit (during bootstrap),
   - Normalize NLU with lookups/synonyms for single-token place names and bus routes,
   - Add a form-on-fallback rule[12] to prevent ask/fallback loops while a slot is requested.
4. Re-run the harness, compare deltas. Stop when the KPIs clear the "passably working" bar (below). Otherwise, return to P2.

Compared to Phase I (train/validate only), Phase II adds *behavioral* criteria measured by the harness:

1. Completion: ≥70% of the 50 scripted tests complete (i.e., the final tracker shows the required slots/actions according to the project schema).
2. Turns: median conversation length ≤ 18 total messages (user+bot), indicating the form submits promptly without extended loops.
3. Help recovery: ≥50% on help-detour tests.
4. Stability constraints: no silent warm-up; no ask/fallback infinite loops; form submit path fires deterministically without a custom action server (utter-only is acceptable at this stage).

These thresholds define a "passably working" assistant in transactional, single-domain settings. Following RASA's guidance to track completion/containment and resolution rates as primary business KPIs for production assistants rather than surface metrics (e.g., conversation length)[13,14], we treat a task-completion rate of ≥70% on scripted flows as a conservative viability target. Industry benchmarks typically regard

[11] Key defaults: `N_TESTS = 50`, `TURN_BUDGET = 20` user replies, `BOT_TURN_CAP = 4` messages captured per bot turn, `REQ_TIMEOUT = 20` s; every fifth test injects a help detour.
[12] A RASA rule that, when a form is active and the NLU predicts `nlu_fallback` (or another low-confidence intent), explicitly routes control back to the same form instead of triggering the global fallback action. Concretely, it is a rule with a condition like `active_loop: schedule_form` and a step intent: `nlu_fallback` followed by action: `schedule_form` (or action: `utter_ask_{requested_slot}`), so that the bot simply re-asks the current slot rather than entering an ask/fallback loop or abandoning the form.
[13] https://rasa.com/blog/how-to-design-chatbot-conversation.
[14] https://rasa.com/blog/conversation-driven-development-iterating-against-kpis.

completion or containment rates in the 60%–70% range as indicative of a useful assistant rather than a proof-of-concept prototype[15]. In our setting, 10–18 turns correspond to roughly one or two clarification cycles beyond the ideal 6–10 turn path needed to collect all slots in *Let's Go*/*MEDIA*, and a help-recovery rate of at least 50% ensures that the `help` path is not a dead end. While the exact numerical values are empirically tuned to these two corpora, the underlying criteria—majority of tasks completed, bounded dialog length, and non-trivial recovery from help—are intended as general, domain-agnostic viability targets for intent-based assistants.

To make the level of intervention in Phase II more concrete, Table 3 quantifies how often each micro-patch type was applied per project, together with the number of schema-validation passes and harness runs required to reach the reported KPIs.

Table 3: Phase II micro-patches and validation/harness loops per project. P_1–P_5 count how often each patch template was applied during the full harness-guided bootstrapping procedure

Model	P_1	P_2	P_3	P_4	P_5	Schema validates	Harness runs
MEDIA (Fr)							
GPT-4o	0	4	0	0	0	2	3
Claude Opus	0	7	0	2	0	2	3
Gemini 2.5 Pro	1	6	0	5	0	5	5
Mistral Small	2	5	3	7	2	6	7
Let's Go (En)							
GPT-4o	2	3	0	2	0	1	2
Claude Opus	2	3	0	5	0	2	3
Gemini 2.5 Pro	3	5	2	3	0	2	4
Mistral Small	0	9	2	7	1	10	8

Note: *Patch types*. P_1: new or corrected form slot mappings (`from_text`/`from_entity`); P_2: rule/form restructuring (atomic edge rules, single governing form, removal of conflicting stories); P_3: replacement of custom actions by utter-only stubs during bootstrap; P_4: NLU normalization via lookups/synonyms for routes, place names, and similar tokens; P_5: form-on-fallback rules that re-ask the requested slot instead of triggering a global fallback. "Schema validates" counts the number of times `rasa data validate -debug` and `rasa train -debug` surfaced non-benign issues that led to fixes in Phase II; "Harness runs" counts how many harness → patch loops were executed per project.

Across all eight projects, 93 micro-patches were applied in total (median ≈11–12 per project). The distribution mirrors the qualitative error taxonomy from Table 1: roughly half of all edits involve rule/form restructuring (P_2; 42 applications) to remove policy conflicts and enforce a single control flow, and about one third target NLU normalization (P_4; 31 applications) for route codes, place names, and similar fragile tokens. Slot-mapping adjustments (P_1; 10 applications) and custom-action stubs (P_3; 7 applications) are less frequent but often high-impact, typically resolving silent slot-filling failures or blocked submits. Form-on-fallback rules (P_5; 3 applications) are rare but decisive when ask/fallback loops occur. The number of schema-validation passes per project is small (1–10), and the harness was usually run 3–8 times before meeting the "passably working" thresholds. In practice, each loop consisted of automatically detecting issues via `rasa data validate` and the harness logs, followed by 1–3 YAML-level edits chosen from the fixed catalogue P_1–P_5, so the manual component is bounded and repeatable rather than open-ended engineering.

[15] https://info.rasa.com/hubfs/PDF_Content/Rasa_Business_Benefits_Tipsheet.pdf.

Concretely, recurring log patterns map almost one-to-one to these templates: for instance, tests that repeatedly ended in `nlu_fallback` while a form was active triggered a P_5 form-on-fallback rule; traces where all slots were filled but the submit action never fired led us to convert post-form stories into explicit submit rules or to simplify competing rules (P_2); and failures restricted to route-code mentions were handled by adding or tightening lookups/regex in the NLU configuration (P_4). As in Phase I, we executed these patches manually, but they are deliberately designed as small, deterministic transformations keyed to specific harness and validator signals, making them amenable to future log-driven "auto-patching" tools.

From a design perspective, this two-phase procedure was chosen over end-to-end fine-tuning or purely prompt-based generation for three reasons. First, it matches how intent-based assistants are engineered in practice: starting from explicit, inspectable artifacts (`domain.yml`, NLU data, rules/forms) and then improving behavior with validation and feedback loops, without requiring large labeled corpora or model retraining[16]. Second, it is LLM- and provider-agnostic: Phase I and Phase II operate on the generated YAML, so the same process applies to proprietary and open-source models and can be ported to other platforms with homologous DSLs. Third, it provides fine-grained diagnosability and control: structural bootstrapping isolates cross-file inconsistencies that a purely prompt-based approach cannot reliably avoid, and the harness exposes concrete failure modes (missing confirmations, loops, dead-end help paths) that would be opaque in an end-to-end fine-tuning setup. In this sense, the two phases are not competing with end-to-end learning, but provide a lightweight, tooling-compatible layer that can sit on top of any LLM generation strategy and support reproducible, production-style quality control.

5 Evaluation Process and Results

This section presents the empirical results and how to read them. We first outline the RASA cross-validation protocol (folds, metrics, unified pipeline) to ensure comparability. We then report aggregate metrics—accuracy and macro-F1 across models and corpora—highlighting ceiling effects and class-imbalance sensitivities. Next, corpus-specific per-intent tables harmonize domain schemas to enable aligned comparisons. We diagnose errors via confusion structure (row-normalized matrices plus top confusion pairs with brief interpretations). Robustness is assessed by linguistic slices (short tokens, temporal/date-ish phrases, amenities, and route codes where applicable), contrasting each slice with its complement. Test-based interaction KPIs—completion, median turns, and help-recovery—link structural fixes to end-to-end behavior. We complement these KPIs with an LLM-as-a-judge evaluation of the scripted Phase-II tests—providing turn-level ratings of task adequacy, naturalness, instruction following, and faithfulness (with inter-rater agreement)—as an offline, human-like quality check, and with a focused human evaluation of the Phase II snapshots by two experts in RASA chatbot development.

5.1 Aggregate Metrics

We have evaluated the four LLM-initialized RASA projects (`gpt4o`, `claude`, `gemini`, `mistral`) using RASA's built-in NLU cross-validation. For each project, we executed:

```
rasa test nlu -cross-validation -folds 5 -out results/...
```

and then used the aggregated `intent_report.json` files to extract *accuracy* and *macro-F1* (macro-averaged F1 over intents). We chose macro-F1 to mitigate class-imbalance effects among short `inform_*` and `request_*` intents. To keep conditions comparable, we used the same DIET-based pipeline for all projects. Phase I scores were obtained on the structurally bootstrapped snapshots (after validation

[16] https://rasa.com/blog/conversation-driven-development-zero-to-one.

but before harness-guided edits), while Phase II scores correspond to the final prototypes used in the interaction harness.

Table 4 summarizes the mean intent classification scores across all projects and phases. From a measurement standpoint, the aggregate NLU metrics are fully automated: the reported accuracy and macro-F1 values are the means over the five folds produced by the standard DIET pipeline without any manual pruning or post-selection. No human annotators or judges are involved at this stage; all labels and predictions are taken directly from the RASA outputs. Because we use a fixed fold count and a single cross-validation run per project, the scores should be interpreted as point estimates conditioned on this protocol rather than as resampled estimates with confidence intervals. Variability across folds is implicitly reflected in the underlying `intent_report.json` files.

Table 4: Cross-validated NLU results before (Phase I) and after (Phase II) harness-guided bootstrapping. Accuracy and Macro-F1 are computed from 5-fold cross-validation with the same DIET pipeline

Corpus	Project	Accuracy		Macro-F1	
		P1	P2	P1	P2
Let's Go					
	GPT-4o	1.000	1.000	1.000	1.000
	Claude Opus	0.697	0.652	0.672	0.643
	Gemini 2.5 Pro	0.697	0.697	0.630	0.630
	Mistral Small	0.624	0.688	0.617	0.648
MEDIA					
	GPT-4o	0.511	0.511	0.462	0.462
	Claude Opus	1.000	1.000	1.000	1.000
	Gemini 2.5 Pro	0.588	0.550	0.501	0.481
	Mistral Small	0.867	0.867	0.861	0.861

Two aspects emerge when comparing Phase I and Phase II. First, Phase I already delivers reasonably strong intent classifiers once structural issues are removed: on *Let's Go*, the models start between 0.62 and 0.67 macro-F1, and on *MEDIA* between 0.46 and 0.86. Second, Phase II has only a modest effect on NLU scores: most rows change by at most 0.02–0.03 macro-F1, with a clear gain for *Let's Go*–Mistral (0.62 → 0.65) and small drifts for *Let's Go*–Claude and *MEDIA*–Gemini. This confirms that the harness-guided loop primarily improves *dialog control* rather than intent recognition, and that the structural bootstrapping in Phase I is sufficient to reach stable NLU performance.

Across LLM families and corpora, the table also clarifies general trends. GPT-4o and Claude reach ceiling performance where their induced schemas are deliberately narrow (two intents on *Let's Go*, two fragments on *MEDIA*), so their perfect macro-F1 should be interpreted as upper bounds conditioned on restricted label sets rather than universal robustness. Gemini and Mistral, which tend to propose broader intent ontologies, achieve solid but non-perfect macro-F1 (0.63–0.65 on *Let's Go*, 0.48–0.86 on *MEDIA*), with the French *MEDIA* corpus generally harder than the English *Let's Go* except for Mistral, whose form- and rule-centric schema stabilizes fragmentary turns. Taken together, Phase I vs. Phase II comparisons show that once cross-file inconsistencies are removed, lightweight DIET pipelines yield reliable intent models, and subsequent improvements in end-to-end behavior stem mainly from Phase II control-flow patches rather than large NLU shifts.

5.2 Per-Intent Corpus-Specific Patterns

To surface where systems diverge at the intent level, we report in Tables 5 and 6 macro-averaged F1 together with per-intent F1 aggregated from `rasa test nlu` cross-validation (CV). We harmonize rows post hoc so that each table covers a representative subset of intents that appears in at least one model's domain schema; a dash (–) means the intent is absent from that model (or not reported by its CV run). Macro-F1 is the unweighted mean over the intents *present for that model* in the table (dashes are excluded from the average).

Table 5: Per-intent F1 on *Let's Go* by phase (aggregate across CV runs). Bold = best per row within each phase

Intent	GPT-4o		Claude		Gemini		Mistral	
	P1	**P2**	**P1**	**P2**	**P1**	**P2**	**P1**	**P2**
Greet	**1.00**	**1.00**	0.76	0.64	–	0.36	–	–
Goodbye	–	–	**0.91**	**0.77**	–	–	0.80	0.67
Help	–	–	–	–	–	**0.91**	–	–
Affirm	–	–	0.62	0.57	–	**1.00**	0.00	0.20
Inform	–	–	–	0.50	0.83	**0.85**	0.47	0.50
Deny	–	–	–	0.60	–	–	0.35	0.00
Request_bus_schedule	**1.00**	**1.00**	0.67	0.76	0.56	0.67	–	0.83
Macro-F1	**1.00**	**1.00**	0.74	0.64	0.70	0.76	0.41	0.44

Table 6: Per-intent F1 on *MEDIA* by phase (aggregate across CV runs). Bold = best per row within each phase

Intent	GPT-4o		Claude		Gemini		Mistral	
	P1	**P2**	**P1**	**P2**	**P1**	**P2**	**P1**	**P2**
Goodbye	**0.67**	0.57	–	–	0.33	0.33	–	–
Request_hotel_search	**0.54**	0.44	–	–	0.43	0.50	–	–
Request_booking — request_hotel_booking	0.00	0.00	–	–	0.31	0.36	**1.00**	0.73
Ask_amenities	0.67	**0.80**	–	–	0.75	0.75	–	–
Check_pets	0.67	**0.80**	–	–	–	–	–	–
Provide_city	0.75	**1.00**	1.00	**1.00**	–	–	–	–
Provide_dates	0.67	0.67	1.00	**1.00**	–	–	–	–
Inform[a]	–	–	–	–	0.85	**0.90**	–	–
Inform_city | inform_neighborhood[b]	–	–	–	–	–	–	0.75	**0.76**
Inform_period[b]	–	–	–	–	–	–	–	**0.53**
Deny	0.57	0.25	–	–	0.40	0.44	–	**0.95**
Affirm	–	–	–	–	–	–	–	**0.92**
Select_hotel	–	–	–	–	**0.46**	0.40	–	–
Macro-F1	0.57	0.57	**1.00**	**1.00**	0.50	0.53	0.88	0.78

Note: [a]Gemini's `inform` is a general slot-fragment intent (e.g., free-form city/dates/constraints). [b]Mistral splits `inform` into finer labels (`inform_city`, `inform_period`) that correspond to location/time fragments.

In these tables:

1. Rows list a harmonized subset of intents that together span conversational edges (e.g., `greet`/`goodbye`), task triggers (e.g., `request_bus_schedule` or `request_hotel_search`), slot fragments (e.g., `inform`/`provide_*`), and decision turns (`affirm`/`deny`).
2. Bold numbers mark the best score *per row* across the four systems.
3. Macro-F1 is computed per model over the intents it actually covers in the table; systems with a narrow domain schema can achieve a high macro-F1 if those few intents are easy.

For each cleaned project, we run `rasa test nlu -cross-validation` and extract per-intent F1 along with macro-F1. Because domain schemas differ across models, we do not rename labels; instead, we report the closest matching intent where applicable and leave truly non-applicable rows as "–". The per-intent tables reuse the same cross-validation runs as the aggregate metrics and simply unpack the fold-aggregated RASA reports at the intent level; again, there is no human adjustment of labels, thresholds, or intent groupings beyond the described harmonization.

Across both corpora, the phase-wise per-intent results show a consistent pattern: Phase I already delivers competitive F1 on frequent, structurally well-defined intents, while Phase II mainly broadens coverage and stabilizes more delicate conversational acts. On *Let's Go*, GPT-4o reaches ceiling on its narrow intent subset in both phases, but the other models illustrate the trade-off more clearly: Claude, Gemini, and Mistral improve on the main task trigger (`request_bus_schedule`) while keeping fragment intents (e.g., `inform`, `help`) strong, with some residual polarity ambiguity on `affirm`/`deny`. Overall, macro-F1 generally increases or remains stable, with Phase II edits redistributing performance toward task-critical labels rather than uniformly lifting all intents.

For *MEDIA*, a similar pattern holds despite the richer morphology and hotel-booking domain. GPT-4o and Gemini generally improve or maintain performance on core slots and task triggers, while Mistral preserves very strong fragment-level and decision-act behaviour in Phase II. Some per-intent F1 scores decrease (e.g., farewells or hotel-selection acts), but these drops are expected side effects of refining label boundaries and broadening coverage rather than optimizing each intent in isolation: newly added or re-specified examples make certain decisions harder but better aligned with the underlying schema. Importantly, macro-F1 is preserved or slightly improved for the broader schemas, whereas Claude remains at ceiling under a deliberately narrow schema.

Taken together, the *Let's Go* (English) and *MEDIA* (French) tables show generalizable trends across LLM families and phases. Phase I plus structural validation is sufficient to obtain solid intent recognizers for high-frequency task and fragment intents, but label coverage is often narrow and some conversational edges remain brittle. Phase II, driven by the harness and micro-patches, tends to (i) increase coverage (more intents present in P2 than P1), (ii) improve or preserve performance on core task intents and slot fragments, and (iii) accept small fluctuations or mild degradation on very sparse, ambiguous, or newly differentiated labels (greetings, goodbyes, polarity, light selection turns). From an ablation perspective, the phase comparison therefore quantifies a shift from "high F1 on a small, easy subset" in Phase I to "broader, more realistic behavior with competitive F1" in Phase II, and this pattern holds across both corpora and all four LLM-initialized projects.

5.3 Confusion Structure

To understand how errors arise, we have examined row-normalized intent confusion matrices for representative systems. The confusion matrices and top confusion pairs are computed from the same cross-validation runs used for NLU evaluation, specifically from the fold-aggregated predictions at the end of Phase

II. For each intent, we normalize the corresponding row over all its test instances and then extract the most frequent misclassified intent pairs. No human relabeling or manual recoding of intents is performed at this stage either; the matrices reflect the raw model behavior under the unified DIET pipeline.

We highlight two projects that are both typical and distinctive in their error profiles: *Let's Go—Claude* and *MEDIA—Gemini*. In each case, we pair the heatmap with the most frequent confusion pairs extracted from the prediction logs. We have selected *Let's Go—Claude* and *MEDIA–Gemini* because they exhibit patterns we repeatedly observed across projects (fragment-task conflation; short-token leakage) while also showing corpus/model-specific signatures (French *réserver/chercher* ambiguity; strong fragment stability vs. brittle decision acts). Quantitatively, in both snapshots more than two thirds of the error probability mass is concentrated in fewer than six intent pairs, and two mechanisms account for most of these errors: fragment-to-task promotion (e.g., `inform` → task intents with row-wise probabilities up to 0.40) and short social tokens drifting among `greet`/`goodbye`/`affirm`/`deny` (individual confusion probabilities typically 0.15–0.45). This concentration suggests that a small number of targeted intent boundaries can recover a large fraction of residual mistakes across corpora.

With regard *Let's Go—Claude*, Fig. 6 shows a compact, conversationally rich domain schema (e.g., `affirm`, `deny`, `greet`, `goodbye`, `inform`, `request_bus_schedule`). Two systematic trends emerge: (i) Fragment-task conflation: `inform` (free-form fragments such as stops/times) is often predicted as the full request `request_bus_schedule` (0.40), and the reverse also occurs to a lesser extent (0.13). This is a classic pressure point when short slot fragments and full task requests share lexical cues. (ii) Polarity & greeting leakage: short tokens ("yes", "no", salutations) cross-confuse (`affirm` ↔`deny` ↔`greet`); e.g., true `affirm` is split across `deny`/`goodbye`/`greet` (0.14 each), and true `deny` leaks into `affirm` (0.20) and `greet` (0.20). `goodbye` is mostly stable (0.83) with occasional attraction to `greet` (0.17).

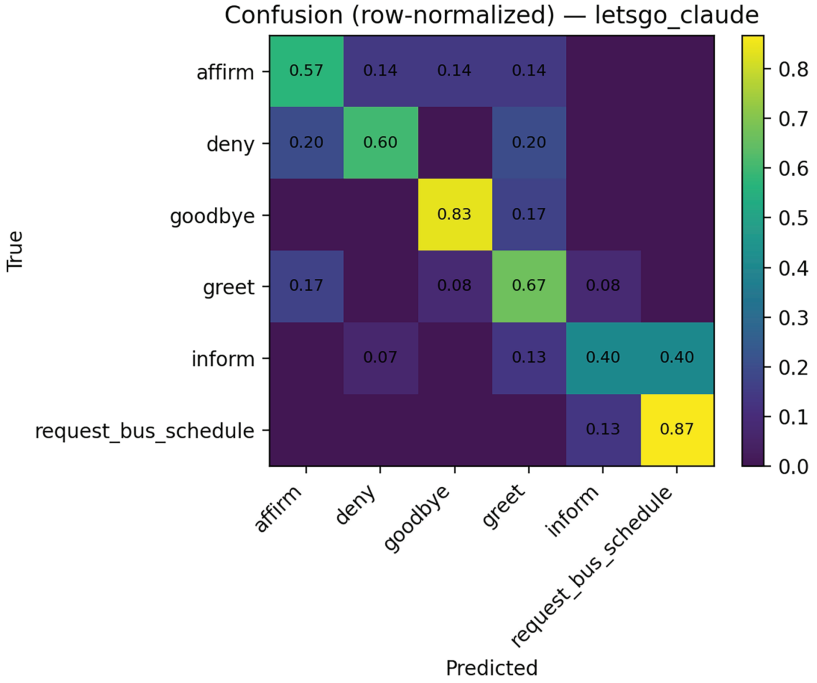

Figure 6: Intent confusion matrix for *Let's Go*—Claude (row-normalized) (Phase II)

Table 7 shows the top-5 confusion pairs for *Let's Go-Claude* (Phase II).

Table 7: Top-5 confusion pairs (true → predicted) for *Let's Go—Claude* (Phase II)

Pair	Relative freq. (%)	Typical trigger
Deny → `inform`	13.3	NO
`inform` → `inform_bus_number`	8.9	54C
`affirm` → `inform`	6.7	SURE
`inform` → `inform_travel_time`	6.7	NOW
`inform` → `inform_departure_stop`	6.7	LEAVING NORTH SIDE

With regard *MEDIA—Gemini*, Fig. 7 includes both *task triggers* (`request_hotel_search`, `request_booking`) and *slot fragments* (`inform`), plus conversational acts (`confirm`, `deny`, `goodbye`) and information-seeking intents (`ask_amenities`, `ask_price`, `select_hotel`). We observe: (i) Near-symmetric task confusions: `request_booking` ↔ `request_hotel_search` (0.67 vs. 0.33) indicates that booking requests are frequently interpreted as fresh searches and vice versa, intuitively plausible given French phrasing where "réserver" often co-occurs with search-like cues. (ii) Goodbye drift into content: `goodbye` leaks notably into `ask_amenities` (0.43) and `inform` (0.29), reflecting short polite endings that borrow lexical material (e.g., "merci") common elsewhere. (iii) Fragments are robust: `inform` is highly stable (0.93), and `ask_price` is strong (0.83). (iv) Light decision acts are brittle: `select_hotel` spreads thinly across neighbors (0.33 correct; 0.17 scattered), suggesting more varied paraphrases would help.

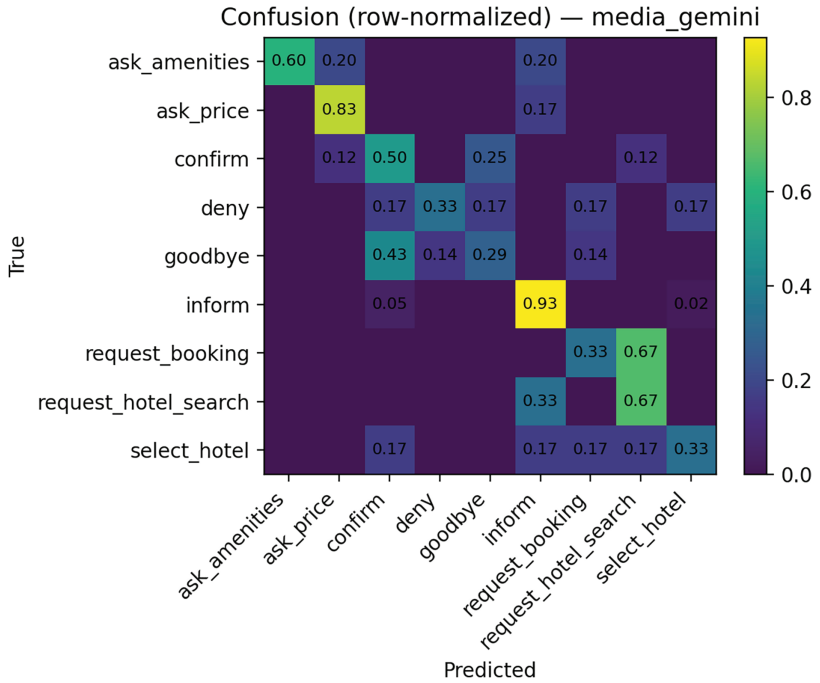

Figure 7: Intent confusion matrix for *MEDIA*—Gemini 2.5 Pro (row-normalized) (Phase II)

Across both matrices, diagonal entries for core task intents remain high (0.67–0.93), whereas conversational edges and fragments exhibit lower row-maxima (typically 0.50–0.83) and broader dispersion. This

pattern indicates that 3–4 conversational and fragment intents per corpus dominate residual confusion and thus are natural targets for focused data augmentation or policy refinement.

Table 8 shows the top-5 confusion pairs for *MEDIA—Gemini*.

Table 8: Top-5 confusion pairs (true → predicted) for *MEDIA*—Gemini 2.5 Pro (Phase II)

Pair	Relative freq. (%)	Typical trigger
`request_booking` → `request_hotel_search`	13.8	je voudrais rèserver alors
`goodbye` → `confirm`	10.3	à bientôt
`confirm` → `goodbye`	6.9	c'est ça
`request_hotel_search` → `inform`	6.9	ce serait pour réserver une semaine
`inform` →`confirm`	6.9	quinze jours

5.4 Robustness by Linguistic Slice

We have also assessed robustness on utterance subsets that commonly degrade intent recognition in deployed systems: (i) Short tokens (SHORT; ≤2 tokens, ASR-like fragments), (ii) Route codes (ROUTE_CODE; e.g., `61A`, `54C`), (iii) Temporal phrases (TEMPORAL; e.g., "tomorrow", "6:30 pm"), and, for *MEDIA*, two domain-focused slices: (iv) Amenity mentions (AMENITY; e.g., *piscine*, *jacuzzi*) and (v) Date-ish expressions (DATEISH; e.g., "en septembre", "du 3 au 5"). Each slice is defined by lightweight rules/regex over the test text; metrics are computed from prediction logs at slice level (macro-F1 within slice). We show separate tables for *Let's Go* and *MEDIA*, omitting slice columns that have no support in that corpus (a dash, "–", means no support for that *model* in the given slice). For each model, we report both the structurally bootstrapped snapshot (Phase I) and the final, harness-stabilized snapshot (Phase II), which allows an ablation-style view of slice robustness across phases.

For the robustness analysis, slice membership is computed in a purely rule-based way (using deterministic token-length and regex criteria), and slice-level macro-F1 is then calculated from the same cross-validation prediction logs used for the aggregate metrics, again with no human filtering or relabeling of utterances. Each slice score therefore reflects a single 5-fold evaluation per project restricted to the examples that match the slice definition, and differences between slices mostly reflect the number and difficulty of examples in that slice rather than measurement noise from repeated runs. We do not report confidence intervals per slice, but the patterns we highlight (e.g., short tokens and date-like expressions as systematic weak points) correspond to broad, corpus-wide trends rather than to isolated folds or outliers.

For the *Let's Go* corpus (Table 9), Phase I already yields reasonably uniform slice performance, and Phase II changes are modest but systematic. GPT-4o remains at ceiling on the slices it covers in both phases, reflecting its narrow evaluated label set. Claude shows slightly lower scores on short and route-code slices after Phase II, but a substantial gain on non-short inputs, indicating that the enriched conversational schema improves non-fragment utterances at some cost on very short ones. Gemini's overall macro-F1 is essentially unchanged, but it clearly improves on short inputs, while its complements remain stable. For Mistral, route codes remain the weakest slice compared to their complement, with short/non-short slices staying in a mid-range band. Overall, Phase II does not radically change slice-level robustness on *Let's Go*, but it redistributes performance between short/route-based inputs and their complements in line with the structural changes applied during harness-guided bootstrapping.

Table 9: Robustness slice F1 on *Let's Go* (columns without corpus support removed). Phase I (P1) vs. Phase II (P2) snapshots for each model. The best results within each column, across models and phases are highlighted in **bold**

Project	Phase	ALL	SHORT	R._C.	TEMP.	NOT_R._C.	NOT_SHORT	NOT_TEMP.
GPT-4o	P1	**1.000**	–	–	**1.000**	**1.000**	–	**1.000**
	P2	**1.000**	–	–	**1.000**	**1.000**	–	**1.000**
Claude Opus	P1	0.672	0.**632**	**0.890**	0.356	0.585	0.494	0.690
	P2	0.611	0.481	0.722	0.323	0.611	**0.899**	0.632
Gemini 2.5 Pro	P1	0.630	0.364	–	–	0.656	0.503	0.650
	P2	0.666	0.618	–	0.733	0.712	0.627	0.640
Mistral Small	P1	0.617	0.451	0.319	–	0.597	0.570	0.617
	P2	0.452	0.345	0.153	–	0.559	0.600	0.604

For the *MEDIA* corpus (Table 10), Phase I and Phase II again show consistent but not dramatic differences. Claude remains a narrow, near-ceiling snapshot in both phases, with strong scores on the slices it covers and no direct support for several difficult slices, so its averages must be read in light of this limited coverage. GPT-4o's slice robustness decreases slightly after Phase II on most reported slices, while complements stay in a similar mid-range band. Gemini, by contrast, improves on most slices (including short inputs) while maintaining a perfect amenity slice across phases. Mistral starts from a strong Phase I profile and becomes more balanced in Phase II: overall performance decreases slightly, but difficult slices such as short and date-like inputs become explicitly supported, with complements remaining high. This suggests better coverage of challenging slices at the cost of a small shift in global averages.

Table 10: Robustness slice F1 on *MEDIA* (columns without corpus support removed). Phase I (P1) vs. Phase II (P2) snapshots for each model. The best results within each column, across models and phases, are highlighted in **bold**

Project	Phase	ALL	SH.	AMEN.	DAT.	NOT_AMEN.	NOT_DAT.	NOT_SH.
Claude Opus	P1	**1.000**	–	–	–	**1.000**	–	**1.000**
	P2	**1.000**	–	–	–	**1.000**	–	**1.000**
Gemini 2.5 Pro	P1	0.501	0.532	**1.000**	0.310	0.473	0.492	0.454
	P2	0.586	0.682	**1.000**	0.322	0.515	0.523	0.476
GPT-4o	P1	0.462	0.487	0.367	0.619	0.421	0.366	0.440
	P2	0.439	0.407	0.367	0.556	0.450	0.411	0.441
Mistral Small	P1	0.861	–	**1.000**	–	0.583	0.861	0.922
	P2	0.806	**0.704**	**1.000**	**0.717**	0.771	**0.794**	0.847

Across models and corpora, the slice-level ablation reveals several general trends. Short, fragmentary inputs are consistently harder than their complements: whenever both SHORT and NOT_SHORT are supported, SHORT F1 is systematically lower or comparable, confirming that minimal context remains a robust challenge. Code-like and temporal expressions are also uneven: in *Let's Go*, bus-route codes and temporal phrases tend to underperform their complements, especially for Mistral (route codes) and Claude (temporal slices), while in *MEDIA*, amenity mentions are reliably easy for systems that model them explicitly, and date-like expressions are noticeably weaker than their non-date complements. Phase II typically changes slice

scores only modestly, with the largest gains on the hardest slices (short and date-like inputs) for Gemini and Mistral, and minor reshaping elsewhere. Overall, these patterns suggest that the two-phase pipeline mainly consolidates robustness on challenging linguistic phenomena rather than radically altering NLU behaviour, and that the observed trends (short tokens and dates as pain points, amenities as comparatively easy) generalize across LLM families and both corpora.

5.5 Test-Based Interaction KPIs

Table 11 summarizes dialog-level performance from our scripted harness (Section 4.4): for each project we run 50 tests, cap per-turn exchanges, and record Completion (share of tests that reach a goal state), Turns (median) from first user message to goal, and Help recovery (share of tests that succeed when a mid-conversation *help* detour is injected).[17]

Table 11: Dialog-level KPIs (median across 50 scripted tests)

Model	Completion (%)		Turns (median)		Help recovery (%)	
	P1	P2	P1	P2	P1	P2
Let's Go (En)						
GPT-4o	32	100	42	10	50	100
Claude Opus	10	100	21	10	0	100
Gemini 2.5 Pro	0	100	21	8	0	100
Mistral Small	0	70	22	12	0	100
MEDIA (Fr)						
GPT-4o	0	100	22	10	0	100
Claude Opus	0	90	21	7	0	100
Gemini 2.5 Pro	0	84	22	6	0	100
Mistral Small	0	70	21	5	0	100

The dialog-level KPIs are likewise fully scripted and free of human subjectivity. We do not randomize tests' order or model settings between runs, so the reported values correspond to a single, reproducible harness pass per snapshot rather than to an average over multiple stochastic simulations. In practice, we found that re-running the same tests yields identical or nearly identical trajectories for a given snapshot (because policies and rules are deterministic), so variability across runs is negligible compared to the systematic Phase I vs. Phase II differences we report.

Before starting Phase II, the same harness was run once per project at the end of Phase I structural bootstrapping; these runs correspond to the P1 columns in Table 11. At that point, all bots could engage in multi-turn task-oriented dialogs (median lengths around 21–22 turns, and 42 for *Let's Go* GPT-4o), but only the *Let's Go* GPT-4o snapshot achieved any substantial task completion (32%, with 50% help recovery). The remaining *Let's Go* and *MEDIA* projects sat at 0% completion and 0% help recovery despite non-trivial dialog lengths, indicating that conversations often stalled in loops (repeated clarification questions, unresolved form submits) or ended prematurely after a `help` request without reaching the goal-state conditions. Phase

[17] In our harness, a conversation is *complete* when project-specific required slots/actions are satisfied; median turns aggregate user and bot messages up to the goal state; "help recovery" is computed only on the subset of tests that include the help detour.

II therefore focused on turning these structurally valid yet behaviorally fragile projects into consistently completing agents by iteratively fixing the harness-visible failure modes.

The main conclusions that can be obtained from the results are:

- **High completion after harness-guided bootstrapping.** In Phase I (P1), only *Let's Go* GPT-4o exceeded 0% completion (32%), whereas all other projects on both corpora failed to reach the goal state at all. After Phase II (P2), all snapshots meet or exceed our "passably working" bar (≥50% completion), with six at ≥84% and four at 100%. Quantitatively, completion rises from 0%–10% to 70%–100%, showing that the harness-guided micro-patches directly translate into large gains in task success.
- **Efficiency tracks structure across phases.** At the end of Phase I, median turns cluster around 21–22 for all models and corpora (with *Let's Go* GPT-4o at 42), confirming that the bots can sustain long dialogs but often wander without reaching a goal. After Phase II, median turns drop to 8–12 on *Let's Go* and 5–10 on *MEDIA* for all projects, consistent with cleaner control flow and earlier form submit. Overall, the P1→P2 deltas show that structural patches shorten dialogs while simultaneously increasing completion, rather than merely "cutting off" difficult conversations.
- **Recovery works when wired.** Help recovery is 100% for every Phase II snapshot, indicating that a simple `help` path (atomic rule → guidance utterance → return to form) is sufficient to robustly re-route tests back to task. In Phase I, all models except *Let's Go* GPT-4o had 0% help recovery (Table 11, P1 column), meaning that a help detour almost never ended in a successful completion. The systematic introduction of "help-and-return-to-form" rules in Phase II is what lifts help recovery from 0% to 100%.
- **Corpus-level patterns across phases.** On *Let's Go*, Phase I harness runs show low completion and long dialogs, with particular fragility around route and time collection. Phase II bootstrapping (forms or atomic rules with explicit slot mappings) lifts GPT-4o, Claude, and Gemini to high completion with short dialogs, and brings Mistral to solid performance, typically missing only late confirmations or route-code details. On *MEDIA*, median dialog lengths are similar across phases, but only Phase II achieves high completion; this indicates that the domain schema was already compact and deterministic, and that the main missing ingredient was a clean control path from city/dates to hotel options and stub booking.
- **Model-specific notes and phase comparison.** After Phase II, GPT-4o performs strongly on both *MEDIA* and *Let's Go*, with high completion and short dialogs. Claude and Gemini already behave reasonably at the end of Phase I and, once patched, reach ceiling or near-ceiling completion with compact trajectories on both corpora, suggesting that most remaining errors stemmed from a few structural glitches rather than NLU limits. Mistral becomes very efficient on *MEDIA* and solid on *Let's Go* in Phase II, whereas Phase I logs show near-zero completion and long, unrecovered dialog s; its main fragilities (route-code handling, polarity flips) are precisely those targeted by the harness-driven micro-patches. Overall, comparing Phase I to Phase II harness runs, small declarative control-flow repairs yield large gains in completion and help recovery across all models and both corpora, while median turns decrease or remain stable even as more tests reach a goal state.

5.6 Offline LLM-as-a-Judge Evaluation of Scripted Phase-II Tests

To complement the engineering-oriented KPIs from the harness (task completion, median turns, help recovery), we conducted an offline conversational quality evaluation on the scripted tests generated at the end of Phase II. For each corpus and each LLM-initialized RASA project we collected the full set of scripted conversations produced by the harness after the last round of Phase II patches. Each system turn in these conversations was then scored by three independent LLM "judges" (GPT-4o-mini, Claude 4.5 Opus, Gemini 2.5 Pro) using a shared rubric.

The rubric asked judges to rate, on a 1–5 Likert scale, four dimensions that approximate typical human conversational judgements: (1) *Task adequacy* (does the reply advance the user's goal?), (2) *Naturalness/politeness*, (3) *Instruction following/non-contradiction* (does the reply comply with the prior system prompts and script without contradicting itself?), and (4) *Faithfulness* (does the reply avoid unsupported claims or actions beyond the given context?). Each judge saw the recent dialog context (up to k turns) and the system's last reply, and returned integer scores plus a short textual justification; no external world knowledge was allowed. We then averaged scores across judges and turns to obtain corpus- and model-level means, and we quantified inter-rater reliability across all rated turns using Cronbach's α and Krippendorff's α (ordinal).

Table 12 summarizes mean scores per corpus and system model. *Faithfulness* is consistently high, indicating that Phase-II assistants rarely hallucinate or leave the scripted context. *Naturalness* reaches solid mid-range values, with *MEDIA* models generally judged more natural than *Let's Go*, plausibly due to the more template-friendly phrasing of hotel-booking dialogs. In contrast, *task adequacy* remains modest, reflecting both the deliberately challenging tests (help detours, partial slot filling, restarts) and the rubric's strict requirement that replies clearly advance the user's goal. *Instruction following* sits in the middle range, suggesting that assistants mostly respect prior commitments but sometimes produce redundant or slightly off-script answers.

Table 12: Offline LLM-as-a-judge quality scores (1–5) on scripted Phase-II tests. Scores are averaged over all rated system turns for each corpus and system model. Higher is better. The best results within each corpus, across models, are highlighted in **bold**

Model	Adequacy	Naturalness	Instruction	Faithfulness
Let's Go (EN)				
Claude 3 Opus	1.95	3.16	2.84	4.51
Gemini 2.5 Pro	**2.12**	**3.38**	**3.04**	**4.67**
GPT- 4o	1.21	1.78	2.24	4.21
Mistral Small	1.84	3.09	2.66	4.28
MEDIA (FR)				
Claude 3 Opus	2.20	3.62	3.03	**4.80**
Gemini 2.5 Pro	**2.42**	**3.88**	**3.08**	4.56
GPT-4o	1.76	3.18	2.64	4.41
Mistral Small	2.30	3.76	2.90	3.73

Across models, Gemini 2.5 Pro and Claude Opus obtain slightly higher adequacy and naturalness, especially on *MEDIA*, while GPT-4o and Mistral Small lag on adequacy for *Let's Go*. This mirrors the harness-based completion scores (Section 5.5): systems with higher task completion on scripted tests also receive higher LLM-judge adequacy and naturalness ratings, indicating that the LLM-as-a-judge setup captures meaningful aspects of task-oriented behavior rather than only surface fluency.

To assess how reliable these LLM-judge scores are as a proxy for human annotation, we computed inter-rater agreement across all rated turns (both corpora and all models). Table 13 reports Cronbach's α (treating the three judge models as raters) and Krippendorff's α for ordinal data. Adequacy and naturalness show substantial internal consistency and moderate agreement, indicating that different LLM judges largely concur on which replies advance the task and sound natural, so averages provide a reasonably stable signal. Instruction following and faithfulness exhibit weaker and more variable agreement, suggesting these

dimensions are harder to judge consistently and should be interpreted as coarse indicators rather than precise rankings. Nonetheless, the consistently high mean faithfulness scores across systems (Table 12) still support the claim that Phase-II bots rarely invent unsupported content on the scripted tests.

Table 13: Inter-rater reliability of LLM judges across all rated system turns, combining both corpora and all four system models. Cronbach's α treats judge models as raters; Krippendorff's α uses squared distances on ordinal scores

Dimension	Cronbach's α	Krippendorff's α (ordinal)
Adequacy	0.787	0.458
Naturalness	0.875	0.510
Instruction	0.508	−0.215
Faithfulness	0.351	0.085

Taken together, these results provide an offline, human-like assessment of the conversational behavior of our LLM-initialized assistants that goes beyond engineering KPIs alone. The scripted tests used by the harness approximate realistic task flows derived from the original corpora, and the LLM-as-a-judge setup yields interpretable judgements for each system turn on task adequacy, naturalness, instruction following, and faithfulness. For each rated turn we obtain three scores per dimension, which allows us to characterize variability across "raters" via Cronbach's and Krippendorff's α rather than through repeated task runs. While this does not replace a full user study with human participants, it offers a scalable proxy that is sensitive to qualitative aspects of interaction and shows non-trivial inter-rater agreement for the most relevant dimensions (adequacy and naturalness).

5.7 Evaluation Completed with Experts

To complement the automatic evaluations, we have conducted a structured assessment with two senior RASA developers, each with more than ten years of experience building task-oriented conversational agents in industrial contexts. Their role was to manually examine the full set of RASA artifacts generated following our proposal (end of Phase II) for the two application domains, the `domain.yml`, `nlu.yml`, `stories.yml`, `rules.yml`, `actions.py`, `config.yml`, and `endpoints.yml` files, and provide an expert judgment of their technical correctness, dialog robustness, and overall suitability for deployment. From an evaluation-methodology perspective, the expert review was carried out once per final Phase-II snapshot. As a result, the human evaluation concerns a single, converged version of each system, making the combination of automatic and expert-based measures easier to interpret.

The evaluation followed a three-stage expert-review protocol. First, an independent file-by-file assessment, in which each expert reviewed all RASA files independently. They evaluated every criterion defined in Table 14 using a Likert scale (1–5, where 5 = optimal quality). Reviewers also noted concrete examples of issues (e.g., invalid slot references, conflicting rules, overly broad intents, etc.). Second, a cross-artifact consistency verification, in which the experts were instructed to analyze not only the internal coherence of each file (e.g., consistency of intents and examples in `nlu.yml`), but also cross-file references. This included checking whether all actions referenced in the stories are implemented in `actions.py`, slots/entities defined in `domain.yml` are used consistently in dialog management files, and the NLU pipeline and policies in `config.yml` are appropriate for the scenario and language. Finally, a consensus building and scoring consolidation after completing their independent scoring, in which the experts compared and

discussed results to align interpretations of the criteria. Minor disagreements (typically ±0.5 rating points) were resolved through discussion, and the final table reports the mean value of the two reviews.

Table 14: Criteria defined for the evaluation with human experts

Category	Criterion	Sub-criterion	Description
I. Consistency and technical correctness	Consistency	**CTC1**. Intent and response usage	All intents and responses in `domain.yml` are referenced in the NLU file and in at least one story or rule.
		CTC2. Reference validity	All slots, entities, and responses referenced in dialog files (`stories.yml`, `rules.yml`) or custom actions are correctly defined in `domain.yml`.
	Dialog structural integrity	**CTC3**. No conflicts	There are no conflicts between stories/rules (e.g., conflicting intent/action sequences).
		CTC4. No dead ends	Stories/Rules have clear end states or fallback mechanisms; they do not terminate abruptly or lead to unhandled states.
	Configuration and deployability	**CTC5**. Pipeline appropriateness	The NLU pipeline is well-suited for the target language, domain vocabulary, and complexity (e.g., using an appropriate featurizer and language model).
		CTC6. Policy selection	The chosen policies (e.g., `TED`, `RulePolicy`) are appropriate for the complexity of the designed dialog flows.
II. NLU & response quality	NLU data quality	**NRQ1**. Intent and entity coverage	All important user intents are covered, including explicitly the relevant entities.
		NRQ2. Intent granularity	Intents are distinct: neither too broad (combining multiple user goals) nor too specific (redundant intents).
		NRQ3. Example diversity	Training examples are diverse and cover various linguistic styles and phrasing for the same intent.
	Response quality	**NRQ4**. Example naturalness	Examples sound like something a real user would genuinely say, not artificially generated.
		NRQ5. Clarity and conciseness	Responses are easy to read and understand.
		NRQ6. Tone and consistency	All responses maintain a consistent and appropriate tone.
III. Dialog flow & task coverage	Core task fulfillment	**DFT1**. Critical flows	All critical tasks are covered by the rules and stories.
		DFT2. Happy paths	Most frequent conversations flow naturally without unnecessary loops or re-prompts.

The evaluation criteria (Table 14) were designed to assess three categories: (i) Consistency and technical correctness: Whether the generated RASA project is structurally valid, internally coherent, and deployable without extensive human rework; (ii) NLU and response quality: the linguistic, semantic, and pragmatic adequacy of the NLU training data and the system responses; and (iii) dialog flow and task coverage: the ability of the generated dialog to support the intended tasks, handle the "happy path", and manage potential

deviations. These categories correspond to known sources of failure in manually and automatically generated RASA systems, such as intent misalignment, incomplete story coverage, or inconsistent action definitions.

Table 15 summarizes the expert scores for both *Let's Go* and *MEDIA*. For consistency and technical correctness (CTC), Claude Opus and Mistral Small are rated highest, with few missing references, coherent stories, and appropriate pipelines and policies. Gemini 2.5 Pro is generally solid but shows occasional missing references or incomplete flows, especially on *Let's Go*, while GPT-4o is the most variable, sometimes over-extending the schema with redundant or unused intents and therefore receiving lower CTC1–CTC2 scores. Across languages, all models produce slightly cleaner, more coherent structures on *MEDIA*, likely reflecting its more constrained task boundaries.

Table 15: Results of the evaluation with human experts

Sub-criterion	Claude opus	Gemini 2.5 pro	GPT-4o	Mistral small
		Let's Go (EN)		
$CTC1$	4	3	3	4
$CTC2$	5	5	2	5
$CTC3$	4.5	3.5	3.5	4.5
$CTC4$	5	3.5	3.5	4.5
$CTC5$	5	4.5	4.5	4.5
$CTC6$	5	5	4.5	5
$NRQ1$	4	3.5	3	4.5
$NRQ2$	5	3.5	4	3.5
$NRQ3$	4	3.5	3	4
$NRQ4$	4	3	4	4
$NRQ5$	5	5	4	5
$NRQ6$	5	5	4.5	5
$DFT1$	4	4	3.5	4.5
$DFT2$	4	4	3	3
		MEDIA (FR)		
$CTC1$	4.5	4	4	4.5
$CTC2$	5	5	4.5	5
$CTC3$	3.5	3.5	4	4
$CTC4$	3	3.5	4	4
$CTC5$	5	5	4.5	5
$CTC6$	5	5	4.5	5
$NRQ1$	4.5	5	3	4
$NRQ2$	5	5	3.5	3
$NRQ3$	3.5	4	3	3.5
$NRQ4$	4.5	4.5	4	4
$NRQ5$	5	5	3.5	5
$NRQ6$	5	5	5	5
$DFT1$	5	4	4	5
$DFT2$	4	4	3.5	4.5

For the NLU-related criteria (NRQ), the results show more variation across LLMs. Claude Opus consistently produced the most diverse and contextually rich examples (NRQ3), with well-separated intent definitions (NRQ2) and high naturalness (NRQ4). Its responses were considered "deployment-ready" without major editing. Gemini 2.5 Pro produced relatively high-quality NLU examples but slightly weaker intent granularity in the Let's Go dataset, occasionally merging distinct user goals into a single intent. GPT-4o produced fluent responses (NRQ4–NRQ6) but demonstrated weaker performance in intent and entity coverage (NRQ1) and example diversity (NRQ3), especially for MEDIA. Experts reported that the generated NLU examples tended to cluster around a single phrasing pattern, which can hinder RASA model performance. Mistral Small scored surprisingly well in response consistency and tone (NRQ6), although its NLU example diversity (NRQ3) was more limited due to shorter and simpler training utterances. Overall, NRQ scores suggest that larger LLMs do better at producing linguistically rich and diverse training data but may still struggle with fine-grained intent differentiation without domain-specific prompting.

The dialog management results (DFT1-DFT2) reveal clear differences across models. Claude Opus and Mistral Small scored highest, indicating they generated stories and rules with strong coverage of critical flows and clean "happy path" execution. Gemini 2.5 Pro performed well but produced slightly more fragmented stories, occasionally requiring manual restructuring. GPT-4o showed the lowest scores in this category for both datasets. Experts observed that GPT-4o tended to generate correct individual stories but lacked global cohesion-certain flows were missing or inconsistent, creating potential dead ends. For the MEDIA dataset, all models showed improvements due to the more procedural and slot-filling nature of the domain, making it easier for LLMs to produce coherent dialog paths.

Beyond the structured scoring presented above, we also collected qualitative feedback from the experts through an open-ended question aimed at understanding their overall perception of the proposal's effectiveness and its practical implications for RASA development workflows. Both experts agreed that the proposed LLM-based framework substantially accelerates the chatbot development, especially in the definition of intents, entities, domain structure, and initial response templates. They highlighted that these components, typically time-consuming during manual creation, were generated with a level of coherence and completeness that considerably reduces human workload. In particular, reviewers emphasized that the proposal was "especially strong in the generation and organization of intents and entities". According to their feedback, the created NLU layer was often sufficiently well-structured to be used as a direct foundation for training, requiring only moderate refinements (e.g., additional examples or small adjustments in labels). This matches the high NRQ scores obtained in the evaluation.

A recurrent theme across the open-ended responses was that dialog management (stories and rules) constitutes the most challenging aspect of RASA development, even for experienced developers. Stories and rules must correctly capture user behavior patterns, domain constraints, and error-handling mechanisms. As described by the experts, in their experience developing RASA chatbots, the most complex part is the development of the dialog flows with stories and rules. They may seem to be valid when reading the `stories.yml` and `rules.yml` files, but the behavior of the system depends heavily on the policies, pipeline, size of the history window, and many other tricky aspects. Also, when the chatbot has a certain size it is not uncommon to find inconsistencies or redundancies in the flows developed, and especial attention must be paid to gracefully handle fallbacks and errors. Thus, in their opinion the results of the automatic generation are in some sense similar to what a RASA developer would produce as a first draft, but would require iterative editing and testing to produce a good result.

The experts also emphasized that the results of automated generation are pretty impressive, especially in putting together different prototype versions of a RASA chatbot. The intents it produces are generally solid, the system prompts are decent too, though a few models returned fewer prompts than they would have expected,

and the generated responses mirror the log text a little too closely. However, what the system generates as stories and rules does not really match what a human RASA developer would develop. Those are usually the hardest and most time-consuming parts of building a bot. Still, having the system auto-generate a first version that the developer would refine and correct could save a significant amount of development time.

6 Discussion

From the evaluation process described in the previous section, we can conclude that LLMs can now draft plausible specifications for RASA task bots, but our eight full builds (4 models × 2 corpora) show that reliable and fully functional assistants assistants still require two disciplined loops: (1) a validator loop that enforces cross-file invariants and policy sanity, and (2) a harness loop that enforces dialogic completeness (first-turn intake, confirmations, recovery, terminalization).

This way, despite producing syntactically plausible RASA artifacts in a single shot, none of the eight projects (4 models × 2 corpora) generate trainable, runnable, and conversation-completing assistants without targeted engineering. Closing the remaining gap likely requires closed-loop generation: generate → validate → auto-patch (e.g., add missing `utter_*`, convert conflicting rules to form prompts) → re-validate → test with harness → auto-patch control stubs → re-test. Lightweight, slice-aware augmentation and language-localized normalizers (reintroduced *after* a clean compile) should further improve robustness—moving closer to one-shot, runnable dialog projects while preserving the rigor needed for reliable behavior across both English (*Let's Go*) and French (*MEDIA*) domains.

Beyond these quantitative results, it is important to clarify how our evaluation relates to classical RASA pipelines and standard NLU models, and why we do not include an additional manually engineered baseline for comparison. Building such a baseline for each corpus would require weeks of expert effort to design the ontology, author and annotate hundreds of training examples, and iteratively debug stories and policies, as documented in prior work on task-based chatbots and RASA-based assistants [16,38,56–60]. Instead, our setup deliberately fixes the NLU and policy backbone to the standard DIET-based RASA configuration and varies only the source of supervision: human-authored domains in the literature vs. LLM-generated schemas and examples in our experiments. In that sense, the DIET models we train already instantiate the classical RASA/standard NLU baseline, and our results should be read as showing that LLM-initialized domains are comparably learnable by this baseline while avoiding the substantial manual design and annotation overhead that a separate handcrafted reference system would entail.

Our approach is intended to be portable across domains and conversational frameworks, but full interoperability still requires some platform-specific adaptation. Different toolchains expose distinct schema formats and control formalisms (e.g., YAML+stories/rules in RASA, JSON+pages/routes in Dialogflow CX, graph/adaptive dialogs in Bot Framework), so induced intents, entities, and abstract dialog transitions must be compiled into platform-native structures, naming conventions, and control units, including any mandated fallbacks or validation turns. NLU engines likewise differ in tokenization, feature extraction, and confidence handling, which typically calls for light rebalancing or simplification of the induced training data, and backend logic must be rewired into platform-native action handlers and augmented with any domain- or regulation-specific constraints. These are engineering-layer transformations: they preserve the underlying induced schema and dialog design while ensuring syntactic and behavioral compatibility when porting the same LLM-initialized assistant across frameworks and deployment contexts.

We discovered also that LLMs are strong at local pattern completion, but reliable global schema discipline remains elusive. Out-of-the-box generations exhibited (a) intent drift (`request_schedule` vs. `request_bus_schedule`), (b) action/utterance drift (stories reference `utter_*` never defined in the domain), and (c) policy contradictions (rules predict a different next action than stories). Our first-phase

bootstrap (validator-led edits, explicit `slot_mappings`, atomic rules or a single form) eliminated compile-time failures, but the second-phase harness still surfaced runtime inconsistencies (e.g., missing confirm prompts or non-deterministic follow-ups) that no static generation reliably solved.

Several models proposed fine-grained, semantically tidy domain schemas (e.g., distinct `provide_departure`, `provide_destination`, `provide_time`). Under tiny data, this dilutes examples per label and raises confusion among short fragments ("*to oakland*", "*6:30 p.m.*"). Cross-validation macro-F1 improved only after either (a) merging fragments implicitly via forms (slot-first control) or (b) augmenting slice-specific paraphrases. The effect is visible in robustness slices: SHORT inputs and code-like tokens depressed F1 unless handled by character n-grams, lookups/regex, or explicit fragment modeling.

The expert evaluation highlights that the proposal is highly effective for automating NLU layers, providing well-defined intents, entities, and example utterances that significantly reduce expert involvement in early configurations. Domain file generation and policy selection are also handled competently, especially by larger models such as Claude Opus. Dialog management remains the hardest component to automate, due to the inherent ambiguity of mapping task flows into RASA stories/rules, the dependency on configuration-based context windows, and the difficulty of ensuring global consistency across branching conversational paths. Human experts still play a key role in validating and refining stories and rules, and this step continues to require substantial time and technical understanding. The experts concluded that the proposal is a strong foundation for a hybrid workflow, where LLMs handle the initial generation and structuring, and developers focus their effort on iterative improvement of the dialog flows—the parts where automatic generation is least reliable.

We can also conclude that syntactically valid YAML does not guarantee a finishable conversation. The harness revealed missing or brittle control points: first-turn activation, turn-taking under slot requests, explicit confirmation prompts, and graceful terminalization. High NLU scores sometimes coexisted with low completion (e.g., *Let's Go* `GPT-4o`: macro-F1 = 1.00 on its narrow label set, yet only 32% completion before harness work), whereas modest NLU could drive strong dialog KPIs once confirm/deny and recovery paths were made deterministic (e.g., *MEDIA* `Mistral`: 100% completion, median 5 turns).

In addition, generated projects frequently assumed components that were not running (e.g., Duckling) or depended on custom actions that stalled the harness environment. We stabilized projects by removing non-running dependencies during bootstrap and, in the harness phase, replacing side-effecting actions with utter-only stubs (e.g., `utter_booking_complete` with static `booking_id`/`total_price` slots) to measure dialog logic independently of servers. Provider constraints also mattered upstream: to generate *MEDIA*, Claude required 10–12 k-token batches due to effective rate limits (despite a nominal 30 k context), Gemini handled 128 k, Mistral 30 k, and GPT-4o ~84 k usable dialog tokens once prompt overhead and counting quirks were accounted for. These constraints shape the breadth and consistency of the initial artifacts.

Given these constraints, we adopted a conservative stance on external parsers such as Duckling in Phase I, ensuring that all projects would validate, train, and run under the harness without any external HTTP services. Slice-level analysis then revealed a clear pattern: temporal and date-like expressions remain the weakest cases across models, whereas route codes and similar tokens are handled reasonably well with character n-grams, simple regexes, and paraphrase augmentation. We therefore treat external extractors as a second-stage enhancement, to be restored only once the LLM-initialized project is structurally clean (Phase I), behaviorally stable under the harness (Phase II), and diagnostics reveal persistent gaps in exactly those entity types (e.g., complex time expressions). At that point, tools like Duckling can be re-enabled behind explicit before/after tests on slice-level F1 and harness KPIs, keeping the integration only if

completion and stability do not regress. For narrow, single-domain assistants such as *Let's Go* and *MEDIA*, our Phase I/II pipeline already yields "passably working" behavior without external services; as domains and temporal/numerical demands grow, restoring external extractors becomes a natural next step that our framework can accommodate and evaluate in a controlled way.

Finally, NLU cross-validation and robustness slices capture recognition quality, but conversation completion depends on control decisions. Our scripted harness tests (median across 50 dialogs) disentangled the two: after the second-phase loop, all projects reached ≥70% completion (many hit 84%–100% with short median turns). The lesson is practical: without deterministic confirm/recovery/terminal behaviors, good NLU does not reliably translate into task success.

7 Conclusions and Future Work

In this paper, we have presented a two-phase method that can be applied to use LLMs to reliably bootstrap task-oriented conversational assistants when guided by systematic validation and minimal structural intervention. Our proposal transforms raw, often inconsistent LLM outputs into executable, coherent dialog agents able of achieving high task-completion rates across distinct corpora and interaction languages. This way, by uniting large-scale language modeling with validator-driven engineering, our proposed approach advances both the automation and the reliability of intent-based conversational AI development without the need for extensive manual tuning or retraining.

The empirical results using the RASA platform and two reference corpora to develop eight system instances confirm that even minimal structural interventions, such as enforcing domain–story–NLU alignment, simplifying dialog policies, and adding explicit slot mappings, yield significant improvements in both robustness and reliability. These findings highlight that the main challenge to practical deployment is not the generative capacity of the models but the lack of post-hoc structural coherence, a gap that validator- and harness-driven mechanisms can systematically close. The performance gains observed after Phase II underscore how modest, rule-based consistency checks can turn preliminary LLM scaffolds into fully functional dialog frameworks with reproducible behavior.

Methodologically, the work contributes a reusable pipeline that operationalizes the transformation from generative artifacts to production-ready conversational projects. Beyond its immediate utility for RASA, the approach is portable to other major intent-based ecosystems, such as Dialogflow, Amazon Lex, or IBM Watson Assistant, given their analogous representations of intents, entities, and dialog control flows. In this sense, the proposed framework provides a blueprint for bridging generative AI and symbolic dialog engineering, suggesting a pragmatic path toward hybrid development workflows that combine scalability with interpretability.

Nevertheless, several limitations remain. The experiments have been constrained to single-domain assistants and two languages, and macro-F1 variations reflect differences in schema scope rather than full generalization. As future work, we want to extend and assess our proposal considering multilingual tasks, multi-domain contexts, and the integration of additional APIs and databases.

Acknowledgement: We thank our colleagues and collaborators for thoughtful feedback on early drafts. Portions of the editing, copy-editing, and language proofreading of this manuscript were assisted by large language models (LLMs), including OpenAI's GPT-5, used under the authors' direction for tasks such as grammar corrections, style harmonization, and clarity improvements. All substantive content, study design, analyses, and final interpretations are the sole responsibility of the authors, who reviewed and approved every revision.

Funding Statement: This publication is part of the TrustBoost project, that has received funding from MICIU/AEI/10.13039/501100011033 and from FEDER, UE. It is a coordinated project by a multidisciplinary team

from the Universidad Politécnica de Madrid (UPM) and University of Granada (UGR), with two subprojects that address TrustBoost's objectives: "Enhancing Trustworthiness in Conversational AI through Multimodal Affective Awareness" (TrustBoost-UPM, ref. PID2023-150584OB-C21), and "Breaking the Duality of Conversational AI: Going beyond Guided Conversations While Ensuring Compliance with Domain Rules and Constraints" (TrustBoost-UGR, ref. PID2023-150584OB-C22).

Author Contributions: The authors confirm contribution to the paper as follows: Conceptualization, Ksenia Kharitonova and David Pérez-Fernández; methodology, David Pérez-Fernández; software, Ksenia Kharitonova; validation, Ksenia Kharitonova; formal analysis, Ksenia Kharitonova, David Griol and Zoraida Callejas; investigation, Ksenia Kharitonova, David Pérez-Fernández, David Griol and Zoraida Callejas; resources, Ksenia Kharitonova; data curation, Ksenia Kharitonova; writing—original draft preparation, Ksenia Kharitonova; writing—review and editing, David Griol and Zoraida Callejas; visualization, Ksenia Kharitonova; supervision, David Pérez-Fernández; project administration, David Griol and Zoraida Callejas; funding acquisition, David Griol and Zoraida Callejas. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: All code, generation scripts and prompts, as well as the harness implementation and the final RASA projects for all eight model-corpus combinations, are publicly available at https://github.com/trustboost-ugr/Logs2Chatbot/. The repository also includes a processed example of the *Let's Go* corpus suitable for use with our scripts. Due to licensing constraints, the *MEDIA* corpus itself is not redistributed and must be obtained separately from ELRA.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

AI	Artificial Intelligence
ASR	Automated Speech Recognition
ATIS	Automatic Terminal Information Service
DIET	Dual Intent and Entity Transformer
DSTC	Dialog State Tracking Challenge
DSL	Domain Specific Language
KPI	Key Performance Indicator
LLM	Large Language Model
OSS	Open Source Software
NLP	Natural Language Processing
NLU	Natural Language Understanding
SLU	Semantic Language Understanding
TPM	Tokens Per Minute
RL	Reinforcement Learning
YAML	Yet Another Markup Language

References

1. McTear M, Ashurkina M. Transforming conversational AI: exploring the power of large language models in interactive conversational agents. Berkeley, CA, USA: Apress; 2024.
2. Deng Y, Liao L, Lei W, Yang GH, Lam W, Chua TS. Proactive conversational AI: a comprehensive survey of advancements and opportunities. ACM Trans Inf Syst. 2025;43(3):67. doi:10.1145/3715097.
3. McTear M, Callejas Z, Griol D. The conversational interface: talking to smart devices. Cham, Switzerland: Springer; 2016.
4. Timpe-Laughlin V, Divekar R, Sydorenko T, Dombi J, Oh S. Intent-based versus GPT-based conversational agents: benefits and challenges for practicing and assessing oral interaction. TESOL Q. 2025;59(S1):S117–49. doi:10.1002/tesq.3409.
5. Zeng Z, Watson W, Cho N, Rahimi S, Reynolds S, Balch T, et al. FlowMind: automatic workflow generation with LLMs. In: Proceeding of the 4th ACM International Conference on AI in Finance (ICAIF '23). New York, NY, USA: ACM; 2023. p. 73–81. doi:10.1145/3604237.3626908.
6. Yu D, Wang M, Cao Y, Shafran I, Shafey L, Soltau H. Unsupervised slot schema induction for task-oriented dialog. In: Proceeding of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Stroudsburg, PA, USA: ACL; 2022. p. 1174–93.
7. Sreedhar MN, Rebedea T, Parisien C. Unsupervised extraction of dialogue policies from conversations. In: Proceeding of the 2024 Conference on Empirical Methods in Natural Language Processing. Stroudsburg, PA, USA: ACL; 2024. p. 19029–45.
8. Chen M, Crook PA, Roller S. Teaching models new APIs: domain-agnostic simulators for task oriented dialogue. arXiv:2110.06905. 2021.
9. Ulmer D, Mansimov E, Lin K, Sun L, Gao X, Zhang Y. Bootstrapping LLM-based task-oriented dialogue agents via self-talk. In: Findings of the association for computational linguistics: ACL 2024. Stroudsburg, PA, USA: ACL; 2024. p. 9500–22. doi:10.18653/v1/2024.findings-acl.566.
10. Kranti C, Hakimov S, Schlangen D. clem: todd: a framework for the systematic benchmarking of LLM-based task-oriented dialogue system realisations. In: Proceeding of the 26th Annual Meeting of the Special Interest Group on Discourse and Dialogue. Stroudsburg, PA, USA: ACL; 2025. p. 62–92.
11. Alhanahnah M, Rashedul Hasan M, Xu L, Bagheri H. An empirical evaluation of pre-trained large language models for repairing declarative formal specifications. Empir Softw Eng. 2025;30(5):149. doi:10.1007/s10664-025-10687-1.
12. Fischer S, Gemmell C, Tecklenburg N, Mackie I, Rossetto F, Dalton J. GRILLBot in practice: lessons and tradeoffs deploying large language models for adaptable conversational task assistants. In: Proceeding of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining; 2024 Aug 25–29; Barcelona, Spain. New York, NY, USA: ACM; 2024. p. 4951–61. doi:10.1145/3637528.3671622.
13. Agrawal S, Pillai P, Uppuluri N, Gangi Reddy R, Li S, Tur G, et al. Dialog flow induction for constrainable LLM-based chatbots. In: Proceeding of the 25th Annual Meeting of the Special Interest Group on Discourse and Dialogue. Stroudsburg, PA, USA: ACL; 2024. p. 66–77.
14. Roy S, Sengupta S, Bonadiman D, Mansour S, Gupta A. FLAP: flow-adhering planning with constrained decoding in LLMs. In: Proceeding of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Stroudsburg, PA, USA: ACL; 2024. p. 517–39.
15. Wang Z, Ribeiro LFR, Papangelis A, Mukherjee R, Wang TY, Zhao X, et al. FANTAstic SEquences and where to find them: faithful and efficient API call generation through state-tracked constrained decoding and reranking. In: Findings of the association for computational linguistics (EMNLP 2024). Stroudsburg, PA, USA: ACL; 2024. p. 6179–91 doi:10.18653/v1/2024.findings-emnlp.359.
16. Bocklisch T, Faulkner J, Pawlowski N, Nichol A. Rasa: open source language understanding and dialogue management. arXiv:1712.05181. 2017.
17. Lison P, Kennington C. OpenDial: a toolkit for developing spoken dialogue systems with probabilistic rules. In: Pradhan S, Apidianaki M, editors. Proceeding of ACL-2016 System Demonstrations. Stroudsburg, PA, USA: ACL; 2016. p. 67–72. doi:10.18653/v1/P16-4012.

18. Pérez-Soler S, Juárez-Puerta S, Guerra E, de Lara J. Choosing a chatbot development tool. IEEE Software. 2021;38(4):94–103. doi:10.1109/MS.2020.3030198.
19. Budzianowski P, Wen TH, Tseng BH, Casanueva I, Ultes S, Ramadan O, et al. MultiWOZ—a large-scale multi-domain wizard-of-Oz dataset for task-oriented dialogue modelling. In: Proceeding of the 2018 Conference on Empirical Methods in Natural Language Processing. Stroudsburg, PA, USA: ACL; 2018. p. 5016–26. doi:10.18653/v1/D18-1547.
20. Sánchez Cuadrado J, Pérez-Soler S, Guerra E, De Lara J. Automating the development of task-oriented LLM-based chatbots. In: Proceeding of the 6th ACM Conference on Conversational User Interfaces; 2024 Jul 8–10; Luxembourg, Luxembourg. New York, NY, USA: ACM; 2024. p. 1–10. doi:10.1145/3640794.3665538.
21. Chatterjee A, Sengupta S. Intent Mining from past conversations for conversational agent. In: Proceeding of the 28th International Conference on Computational Linguistics. Stroudsburg, PA, USA: ACL; 2020. p. 4140–52. doi:10.18653/v1/2020.coling-main.366.
22. Alexiadis A, Nizamis A, Koskinas I, Ioannidis D, Votis K, Tzovaras D. Applying an intelligent personal agent on a smart home using a novel dialogue generator. In: Artificial intelligence applications and innovations (AIAI 2020). Cham, Switzerland: Springer; 2020. p. 384–95. doi:10.1007/978-3-030-49186-4_32.
23. Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, et al. Language models are few-shot learners. In: Proceeding of the 34th International Conference on Neural Information Processing Systems. Vol. 33. Red Hook, NY, USA: Curran Associates Inc.; 2020. p. 1877–901.
24. OpenAI, Achiam J, Adler S, Agarwal S, Ahmad L, Akkaya I, et al. GPT-4 technical report. arXiv:2303.08774. 2024.
25. Priyanshu A, Maurya Y, Hong Z. AI governance and accountability: an analysis of anthropic's claude. arXiv:2407.01557. 2024.
26. Zamfirescu-Pereira JD, Wei H, Xiao A, Gu K, Jung G, Lee MG, et al. Herding AI cats: lessons from designing a chatbot by prompting GPT-3. In: Proceeding of the ACM Designing Interactive Systems Conference. New York, NY, USA: ACM; 2023. p. 2206–20. doi:10.1145/3563657.3596138.
27. Zamfirescu-Pereira JD, Wong RY, Hartmann B, Yang Q. Why johnny can't prompt: how non-AI experts try (and Fail) to design LLM Prompts. In: Proceeding of the Conference on Human Factors in Computing Systems. New York, NY, USA: ACM; 2023. p. 1–21. doi:10.1145/3544548.3581388.
28. Laban P, Hayashi H, Zhou Y, Neville J. LLMs get lost in multi-turn conversation. arXiv:2505.06120. 2025.
29. Bocklisch T, Werkmeister T, Varshneya D, Nichol A. Task-oriented dialogue with in-context learning. arXiv:2402.12234. 2024.
30. Du Y, Wang B, He Y, Liang B, Wang B, Li Z, et al. MemGuide: intent-driven memory selection for goal-oriented multi-session LLM agents. arXiv:2505.20231. 2025.
31. Joshi H, Liu S, Chen J, Weigle L, Lam M. Controllable and reliable knowledge-intensive task-oriented conversational agents with declarative genie worksheets. In: Proceeding of the 63rd Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA, USA: ACL; 2025. p. 27264–308.
32. Uprety SP, Jeong SR. The impact of semi-supervised learning on the performance of intelligent chatbot system. Comput Mater Contin. 2022;71(2):3937–52. doi:10.32604/cmc.2022.023127.
33. Martínez-Gárate A, Aguilar-Calderón JA, Tripp-Barba C, Zaldívar-Colado A. Enhancing conversational agent development through a semi-automatization development proposal. Appl Sci. 2025;15(3):1139. doi:10.3390/app15031139.
34. Samarinas C, Promthaw P, Nijasure A, Zeng H, Killingback J, Zamani H. Simulating task-oriented dialogues with state transition graphs and large language models. arXiv:2404.14772. 2024.
35. Young S, Gašić M, Thomson B, Williams JD. POMDP-based statistical spoken dialog systems: a review. Proc IEEE. 2013;101(5):1160–79. doi:10.1109/JPROC.2012.2225812.
36. Weld H, Huang X, Long S, Poon J, Han SC. A survey of joint intent detection and slot filling models in natural language understanding. ACM Comput Surv. 2022;55(8):156. doi:10.1145/3547138.
37. Cañas P, Griol D, Callejas Z. Towards versatile conversations with data-driven dialog management and its integration in commercial platforms. J Comput Sci. 2021;55(1):101443. doi:10.1016/j.jocs.2021.101443.
38. Vlasov V, Mosig JEM, Nichol A. Dialogue transformers. arXiv:1910.00486. 2020.

39. Henderson M, Thomson B, Young S. Word-based dialog state tracking with recurrent neural networks. In: Proceeding of the 15th Annual Meeting of the Special Interest Group on Discourse and Dialogue (SIGDIAL). Stroudsburg, PA, USA: ACL; 2014. p. 292–9.
40. Mesnil G, Dauphin Y, Yao K, Bengio Y, Deng L, Hakkani-Tur D, et al. Using recurrent neural networks for slot filling in spoken language understanding. IEEE/ACM Trans Audio Speech Lang Process. 2015;23(3):530–9. doi:10.1109/TASLP.2014.2383614.
41. Levin E, Pieraccini R, Eckert W. A stochastic model of human-machine interaction for learning dialog strategies. IEEE Trans Speech Audio Process. 2000;8(1):11–23. doi:10.1109/89.817450.
42. Williams JD, Asadi K, Zweig G. Hybrid Code Networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. In: Proceeding of the 55th Annual Meeting of the Association for Computational Linguistics. Stroudsburg, PA, USA: ACL; 2017. p. 665–77.
43. Williams J, Raux A, Ramachandran D, Black A. The dialog state tracking challenge. In: Proceeding of the SIGDIAL, 2013 Conference. Stroudsburg, PA, USA: ACL; 2013. p. 404–13.
44. Tur G, De Mori R. Spoken language understanding: systems for extracting semantic information from speech. Hoboken, NJ, USA: Wiley; 2011. doi:10.1002/9781119992691.
45. Hakkani-Tür D, Tur G, Celikyilmaz A, Chen Y, Gao J, Deng L, et al. Multi-domain joint semantic frame parsing using bi-directional RNN-LSTM. In: Proceeding of Interspeech 2016. Rennes Cedex, France: International Speech Communication Association; 2016. p. 715–9. doi:10.21437/Interspeech.2016-402.
46. Kvale K, Sell OA, Hodnebrog S, Følstad A. Improving conversations: lessons learnt from manual analysis of chatbot dialogues. In: Chatbot research and design (CONVERSATIONS 2019). Cham, Switzerland: Springer; 2020. p. 187–200. doi:10.1007/978-3-030-39540-7_13.
47. Schmitt A, Ultes S, Minker W. A parameterized and annotated spoken dialog corpus of the CMU let's go bus information system. In: Proceeding of the 8th International Conference on Language Resources and Evaluation (LREC'12). Stroudsburg, PA, USA: ACL; 2012. p. 3369–73.
48. Ultes S, Platero Sánchez MJ, Schmitt A, Minker W. Analysis of an extended interaction quality corpus. In: Lee GG, Kim HK, Jeong M, Kim JH, editors. Natural language dialog systems and intelligent assistants. Cham, Switzerland: Springer International Publishing; 2015. p. 41–52. doi:10.1007/978-3-319-19291-8_4.
49. Shen Z, Wang DYB, Mishra SS, Xu Z, Teng Y, Ding H. SLOT: structuring the output of large language models. In: Proceeding of the Conference on Empirical Methods in Natural Language Processing. Stroudsburg, PA, USA: ACL; 2025. p. 472–91.
50. Meeus Q, Moens MF, hamme Van H. MSNER: a multilingual speech dataset for named entity recognition. In: Proceeding of the 20th Joint ACL-ISO Workshop on Interoperable Semantic Annotation (LREC-COLING). Stroudsburg, PA, USA: ACL; 2024. p. 8–16.
51. Laperrière G, Ghannay S, Jabaian B, Estève Y. A dual task learning approach to fine-tune a multilingual semantic speech encoder for spoken language understanding. In: Proceeding of Interspeech 2024. Rennes Cedex, France: International Speech Communication Association; 2024. p. 812–6.
52. Hurst A, Lerer A, Goucher AP, Perelman A, Ramesh A, Clark A, et al. GPT-4o system card. arXiv:2410.21276. 2024.
53. Anthropic. The Claude 3 Model Family: Opus, Sonnet, Haiku. Anthropic PBC; 2024. [cited 2025 Dec 20]. Available from: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf.
54. Comanici G, Bieber E, Schaekermann M, Pasupat I, Sachdeva N, Dhillon I, et al. Gemini 2.5: pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. arXiv:2507.06261. 2025.
55. MistralAI. Mistral Small 3.1. Mistral AI; 2025 [cited 2025 Dec 20]. Available from: https://mistral.ai/news/mistral-small-3-1.
56. Lam KN, Le NN, Kalita J. Building a Chatbot on a closed domain using RASA. In: Proceeding of the 4th International Conference on Natural Language Processing and Information Retrieval. New York, NY, USA: ACM; 2021. p. 144–8.

57. Paranjape A, Patwardhan Y, Deshpande V, Darp A, Jagdale J. Voice-based smart assistant system for vehicles using RASA. arXiv:2312.01642. 2023.
58. Clerissi D, Masserini E, Micucci D, Mariani L. Towards multi-platform mutation testing of task-based chatbots. In: Proceeding of 36th International Symposium on Software Reliability Engineering Workshops (ISSREW); 2025 Oct 21; São Paulo, Brazil. p. 205–8. doi:10.1109/ISSREW67781.2025.00073.
59. Gatti A, Mascardi V, Ferrando A. RV4Chatbot: are chatbots allowed to dream of electric sheep? Electron Proc Theor Comput Sci. 2024;411(3):73–90. doi:10.4204/EPTCS.411.5.
60. Abdellatif A, Badran K, Costa DE, Shihab E. A transformer-based approach for augmenting software engineering chatbots datasets. In: Proceeding of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. New York, NY, USA: ACM; 2024. p. 359–70. doi:10.1145/3674805.3686695.