SDialog: A Python Toolkit for End-to-End Agent Building, User Simulation, Dialog Generation, and Evaluation

Sergio Burdisso* [1] **Séverin Baroudi*** [1, 2] **Yanis Labrak*** [1, 3]
David Grunert [5] **Pawel Cyrta** [6] **Yiyang Chen** [5] **Srikanth Madikeri** [5]
Esaú Villatoro-Tello [1] **Ricard Marxer** [2, 7] **Petr Motlicek** [1]

[1]Idiap Research Institute [2]Université de Toulon, Aix Marseille Univ, LIS
[3]Avignon University [5]University of Zurich [6]Stenograf [7]ILLS, CNRS

sergio.burdisso@idiap.ch

Abstract

We present SDialog, an MIT-licensed open-source Python toolkit for end-to-end development, simulation, evaluation, and analysis of LLM-based conversational agents. Built around a standardized Dialog representation, SDialog unifies persona-driven multi-agent simulation with composable orchestration for controlled synthetic dialog generation; multi-layer evaluation combining linguistic metrics, LLM-as-a-judge assessments, and functional correctness validators; mechanistic interpretability tools for activation inspection and causal behavior steering via feature ablation and induction; and audio rendering with full acoustic simulation, including 3D room modeling and microphone effects. The toolkit integrates with major LLM backends under a consistent API, enabling mixed-backend and reproducible experiments. By bridging agent construction, user simulation, dialog generation, evaluation, and interpretability within a single coherent workflow, SDialog enables more controlled, transparent, and systematic research on conversational systems.[1]

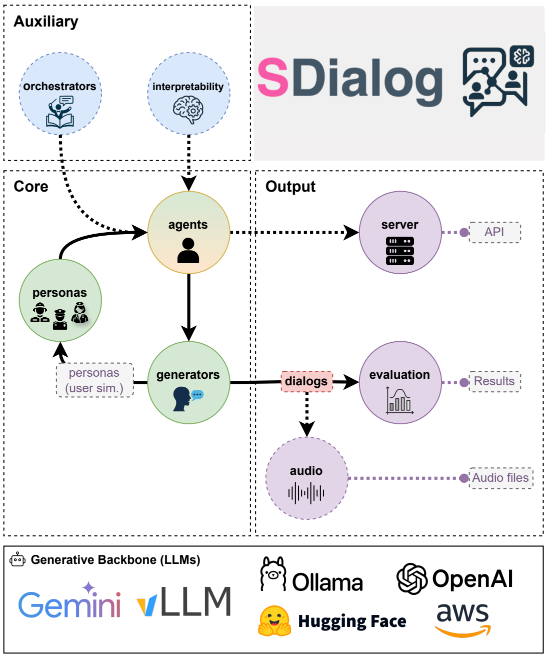

Figure 1: SDialog architecture overview showing eight modules organized into auxiliary, core, and output components.

1 Introduction

The rapid advancement of large language models (LLMs) has enabled increasingly sophisticated conversational AI agents (OpenAI, 2023). Yet despite these gains, researchers lack an integrated and reproducible toolkit for building, controlling, evaluating, and analyzing dialog systems. Current workflows remain fragmented: dialog datasets use inconsistent formats; synthetic data generation tools offer limited control; evaluation practices vary widely across studies; and there is little support for understanding the internal mechanisms that govern model behavior. These gaps hinder progress toward developing robust, transparent, and reproducible conversational systems.

Early efforts such as persona-based generation (Zhang et al., 2018) and infrastructures like RASA (Bocklisch et al., 2017) for building production-level chatbots or ParlAI (Miller et al., 2017) for model training provide useful mechanisms for data handling and dialog management, yet they offer limited support for fine-grained LLM-based dialog orchestration or behavior analysis. More recent multi-agent frameworks such as AutoGen (Wu et al., 2024), AutoGen Studio (Dibia et al., 2024) and smolagents (Roucher et al., 2025) enable dynamic synthetic data creation, but their conversational autonomy often introduces nondeterminism,

[1] **Github:** https://github.com/idiap/sdialog
Demo video: https://youtu.be/oG_jJuU255I
*Main authors

Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics
Volume 3: System Demonstrations, pages 320–340
March 24-29, 2026 ©2026 Association for Computational Linguistics

making it difficult to run controlled experiments or reproduce outcomes. Overall, existing tools focus primarily on data creation while lacking comprehensive evaluation capabilities and mechanisms for interrogating or steering internal model behavior (Caldarini et al., 2022; Singh and Namin, 2025).

These limitations extend to evaluation practices. LLM-based evaluation methods like G-EVAL (Liu et al., 2023) and ChatEval (Chan et al., 2024) better align with human judgments (Li et al., 2025), yet they remain output-focused and provide no insight into why a dialog agent behaves as it does. As a result, evaluation remains decoupled from model introspection, limiting the development of more interpretable and controllable dialog systems.

Meanwhile, advances in mechanistic interpretability (MI) have demonstrated the potential to analyze and influence LLM behavior (Zou et al., 2023a; Arditi et al., 2024a). However, these techniques remain largely disconnected from dialog-centric workflows. Integrating MI into dialog tooling is essential: the ability to inspect internal activations, manipulate high-level behavioral attributes, or enforce desired conversational traits could substantially improve controllability, evaluation fidelity, and system transparency. Yet, to the best of our knowledge, TransformerLens (Nanda and Bloom, 2022) and other MI libraries are not designed around dialogs.

To address these challenges, we introduce `SDialog`, a toolkit that unifies these fragmented workflows into a single, coherent system articulated around the `Dialog` class (§2), while providing an integrated platform for synthetic dialog generation, comprehensive evaluation, user simulation and mechanistic interpretability (§3).

2 A `Dialog` –Centric Architecture

As illustrated in Figure 1, SDialog's architecture is organized around a central `Dialog` object (§2), which serves as the common representation connecting modules for persona-driven generation, orchestration, evaluation, mechanistic interpretability and audio generation (§3). This structure enables a seamless pipeline: agents create `Dialog`s under the guidance of orchestrators, evaluation tools then assess their quality, and interpretability hooks inspect the model behavior that produced them.

In this context, the `Dialog` object serves as the core abstraction. Dialogs are rich objects containing an ordered list of turn instances (speaker and text), optional event objects for internal actions (thinking, tool calls, orchestration) and comprehensive metadata for reproducibility (version, timestamp, model, seed, context, personas, lineage tracking, etc.), that can be created, loaded, transformed, saved and evaluated.[2]

This dialog-centric design enables seamless workflows from generation to evaluation with full provenance. Humans or persona-driven agents generate `Dialog` objects. This architecture unifies previously disconnected components of the dialog research ecosystem, accelerating progress toward transparent and controllable conversational systems.

Multi-Backend Support. To ensure broad applicability, SDialog abstracts LLM interactions through a unified configuration layer. This supports major backends, including OpenAI, vLLM, HuggingFace Transformers, Ollama, Google Gemini and AWS Bedrock, allowing any component such as agent, generator or evaluator, to use different models with fine-grained control while maintaining a consistent workflow.

3 Main Modules

3.1 `personas` Module

This module defines structured personas that drive role-play for user simulation and synthetic dialog generation. Personas are Python classes inheriting from `BasePersona`, which supports attribute introspection, JSON serialization, prompt generation, cloning with lineage tracking, and file I/O. SDialog provides a generic `Persona` and 30+ specialized classes (e.g., `Customer`, `SupportAgent`, `Teacher`, `Student`, `Nurse`, etc.). Users can create custom personas by subclassing `BasePersona` and declaring domain-specific typed fields. An example support agent persona (class `SupportAgent`) is shown in §A.2.

3.2 `agents` Module

This module contains classes for LLM-backed conversational actors. The `Agent` class encapsulates a persona together with conversation memory, optional function-calling tools, orchestration pipelines, and interpretability hooks. It supports configurable first utterances, a "thinking mode" for capturing hidden reasoning, and pre/post-processing hooks for text normalization.

[2] An example dialog JSON object can be found here.

A core capability is dialogue generation: calling `agent_a.dialog_with(agent_b)` produces a complete `Dialog` object (see §A.4 for a concrete example). Generated dialogues serve two primary purposes: (1) evaluating conversational systems by analyzing agent responses and tool usage, and (2) creating synthetic dialogue datasets for downstream uses such as model training, benchmarking, and fine-tuning. Agents can also be served as OpenAI-compatible REST endpoints for live interaction (implementation example in §A.2), or wrapped around existing OpenAI-compatible APIs to proxy external systems for evaluation with simulated users.

3.3 `orchestrators` Module

Orchestrators dynamically control agent behavior by monitoring dialog state and injecting instructions when specific events occur or constraints are satisfied. Instructions can be ephemeral (one-time) or persistent (multi-turn). Built-in orchestrators include: trigger-based instruction injection, conversation length constraints, probabilistic opinion revision, semantic response suggestions, and deterministic scripted sequences. Multiple orchestrators can be composed via the pipe operator, as in the following example:

```python
from sdialog.orchestrators import LengthOrchestrator, SimpleReflexOrchestrator
# Instantiate orchestrators to:
# 1. Keep dialog within 8-12 turns
len_orch = LengthOrchestrator(min=8,max=12)
# 2. Inject instructions on conditions
reflex_orch = SimpleReflexOrchestrator(
  condition=lambda utt: "confused" in utt,
  instruction="Be brief; add an example."
)
# Compose orchestrators with the agent
agent = agent | len_orch | reflex_orch
```

Custom orchestrators can be easily created by inheriting from `BaseOrchestrator`.

3.4 `generators` Module

This module provides a unified, controllable pipeline for creating and transforming conversational data with concrete, easy-to-use classes. At the attribute level, `PersonaGenerator` and `ContextGenerator` build structured personas and contexts using hybrid rules (ranges, files, callables) combined with LLM guidance to balance determinism and diversity. In our use case evaluation, we use `PersonaGenerator` to create the simulated customer personas that interact with the support agent (see §A.3). At the dialog level, `DialogGenerator` creates multi-turn conversations from free-form instructions, while `PersonaDialogGenerator` orchestrates interactions between persona- or agent-based actors to ensure consistent characterization and tool usage. For transformation, `Paraphraser` rewrites existing dialogs (e.g., tone, style, simplification) while preserving speaker identity. All generators track provenance and offer reproducible I/O, enabling systematic dataset creation and fair model comparisons.

3.5 `evaluation` Module

This module provides comprehensive dialog assessment capabilities organized into three layers: individual dialog metrics, dataset-level evaluators, and cross-dataset comparison.

Dialog Metrics Dialog metrics assess individual conversations and return numerical scores or structured outputs. All metric classes inherit from `BaseDialogScore`, which users can extend to implement custom evaluation criteria. SDialog includes diverse built-in metrics organized into six categories:

- *Conversational Features*: Structural and interaction metrics—mean turn length, turn-taking balance, hesitation/question rates, lexical diversity (type–token ratio), back-channel frequency, filler density.
- *Readability Metrics*: Text complexity measures including Gunning Fog, Flesch Reading Ease, Coleman-Liau Index , Linsear Write, and Dale-Chall.
- *Embedding-Based Metrics*: Semantic similarity assessment using neural sentence encoders to compute distances between dialogs or against reference distributions in embedding space.
- *LLM-as-a-Judge*: Prompted LLM evaluators with Jinja2 templates for binary or scalar scoring; built-ins cover realism, refusal detection, persona adherence, optionally returning rationale.
- *Flow-Based Metrics*: Graph-theoretic coherence measures based on dialog flow patterns. These metrics construct probabilistic graphs from reference dialogs where nodes represent semantically similar utterance clusters and edges encode transition likelihoods (Burdisso et al., 2024).
- *Functional Correctness*: Validators for tool-using agents that verify correct behavior in function-calling scenarios, including checking whether tool

invocations follow required sequences (e.g., authentication before data access).

A concrete example use of an LLM-as-Judge and a functional correctness metric are given in §A.5.

Dataset Evaluators Dataset evaluators aggregate individual dialog scores to assess entire collections. Built-in evaluators include: distributional statistics (mean, standard deviation, min, max, median), frequency counting (proportion of dialogs meeting a condition), kernel density divergence for distribution comparison, Fréchet distance between score or embedding distributions (Xiang et al., 2021), and precision-recall curves for embedding space analysis (Xiang et al., 2021). Users can define custom dataset evaluators by inheriting from `BaseDatasetScoreEvaluator`.

Dataset Comparator The `Comparator` orchestrates multi-evaluator, multi-dataset experiments. It accepts a list of evaluators, applies them to multiple named datasets, and generates comparative visualizations via `plot()`. This facilitates systematic benchmarking: e.g., comparing realism rates, readability scores, and flow coherence across different model sizes or agent designs. Complete usage is illustrated in §A.5.

3.6 `interpretability` Module

This module enables interpretability of LLM behaviors through activation capture and steering capabilities, designed specifically for dialog workflows.

Activation Inspection The `Inspector` class attaches PyTorch (Paszke et al., 2019) forward hooks to specified model layers, capturing per-token activations during generation, at turn-level and token-level. It supports monitoring of multiple target layers and provides utilities to influence and control model behaviors. Inspectors can be seamlessly attached to an agent via the pipe operator (§C.2):

```python
inspector = Inspector('model.layers.15')
agent = agent | inspector
agent("How are you?")  # I'm doing great!
agent("That's great!")  # Thanks! I’m glad
# Access last-response first-token activ.
act = inspector[-1][0].act
```

Activation Steering The `Inspector` class supports activation manipulation to causally alter the agent responses. Given a target activation $\mathbf{x}$, behaviors can be suppressed through feature ablation, implemented via the subtraction operator:

$$\mathbf{x}' \leftarrow \mathbf{x} - \hat{\mathbf{r}}\hat{\mathbf{r}}^\top \mathbf{x} \tag{1}$$

where $\hat{\mathbf{r}}$ is a normalized steering vector. Thanks to SDialog, this operation naturally translates to (§C.4):

```python
agent = agent | inspector_x - r  # Ablate
```

For example, if r is a refusal direction, we can prevent the agent from refusing, after the above code:

```python
print(agent("How to make a bomb ?"))
# "To make a bomb, you need (...)"
```

Refer to §C for a detailed case study of the *refusal direction* (Arditi et al., 2024b) using SDialog. Conversely, behaviors can be induced through feature induction using the addition operator (§C.5):

$$\mathbf{x}' \leftarrow \mathbf{x} + \mathbf{r} \tag{2}$$

Similarly, feature induction is expressed intuitively:

```python
agent = agent | inspector_x + r  # Induce
```

Custom steering functions can be defined by subclassing `DirectionSteerer`, and the seamless integration of the `interpretability` module into SDialog enables powerful combinations such as conditional steering when `Inspectors` are combined with `Orchestrators`.

3.7 `audio` Module

This module enables the conversion of dialog objects into synthetic audio datasets, facilitating the generation of realistic spoken dialog corpora for training and evaluation of speech-based systems with simulated physical environment. The conversion process operates through Text-to-Speech (TTS) synthesis followed by acoustic simulation, as follows:

```python
audio_dialog = dialog.to_audio(
    perform_room_acoustics=True
)
```

Text-to-Speech (§B.1): The audio generation process is managed by the `AudioDialog` class, which extends the core `Dialog` data structure. The system utilizes a modular TTS architecture that supports multiple backends through a common `BaseTTS` interface. Voice assignment can be automated via voice databases that map persona attributes, such as age, gender, and language to specific voices.

Acoustic Simulation SDialog can render dialogs within simulated 3D acoustic environments. This process is separated into two main stages: environment definition and audio rendering.

We start by defining a `Room` object (§B.2) for the scene's geometry and acoustic properties by specifying dimensions and surface materials with corresponding absorption coefficients. SDialog provides procedural generators, which can create pre-configured layouts. Audio sources (speakers) and receivers (microphones) are then positioned at specific 3D coordinates within this room (§B.3).

The audio is rendered using a combination of two libraries. dScaper (Grünert et al., 2025) is used to organize all acoustic events (e.g., utterances, background noise) into a spatio-temporal timeline (§B.4). This timeline is then processed by pyroomacoustics (Scheibler et al., 2018), which simulates the sound propagation, modeling reflections via image source methods or ray tracing, and accounting for frequency-dependent air absorption. The sound quality of the recording devices is also simulated by applying a convolution with the impulse response of selected microphones (§B.5). Impulse response databases contains measurements from various physical microphones, enabling the simulation of their distinct frequency responses and characteristics.

4 Use Case Evaluation

We evaluate SDialog by illustrating its end-to-end workflow capabilities through a concrete call-center scenario that exercises the complete pipeline—agent construction, user simulation, dialog generation, and multi-metric evaluation. As an illustrative research question, we compare Qwen3 model sizes (0.6B, 1.7B, 8B, 14B) for their balance of functional correctness and linguistic accessibility. While simplified for clarity, the same workflow generalizes to comparing alternative agent designs (different prompts, tools, orchestrators) or evaluation criteria. The complete evaluation workflow with full implementation details is provided in §A.

The evaluation exercises four key capabilities: (1) rapid agent prototyping with personas and tools, (2) systematic persona variation through `PersonaGenerator`'s flexible attribute rules, (3) mixed-backend support for comparing local models while using more capable models for auxiliary tasks, and (4) multi-dimensional assessment through composable evaluators combining LLM judges, programmatic validators, and linguistic metrics.

Model	**Case A (Verification Required)**		**Case B (No Verification)**	
	Ask-Verify	Tools-OK	Ask-Verify	Tools-OK
qwen3:0.6b	0.82	0.01	0.63	0.09
qwen3:1.7b	0.33	0.00	0.18	0.00
qwen3:8b	0.97	**0.83**	0.38	0.82
qwen3:14b	**1.00**	0.56	**0.06**	**0.93**

Table 1: Functional correctness across Qwen3 sizes. Metrics show proportion of dialogs where agent asks for verification (Ask-Verify) and correctly follows excepted tool sequences in each case (Tools-OK).

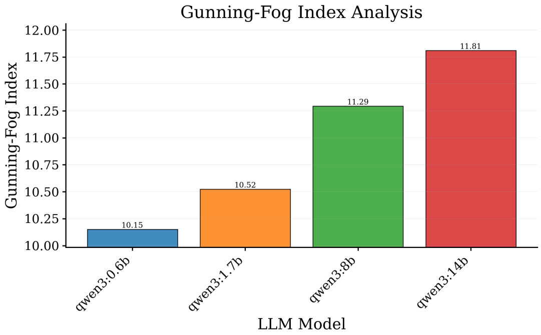

Figure 2: Average Gunning Fog scores increase with model size, indicating more complex language in larger models.

4.1 Workflow Implementation

We demonstrate each workflow stage using SDialog's components.

(1) Backend Configuration (§A.1): SDialog's multi-backend support allows mixing model sources. We configured Ollama for local Qwen3 models (evaluation targets) while using OpenAI GPT-4.1 for auxiliary components (customer simulation and LLM-as-a-judge evaluators). This illustrates SDialog's flexibility: practitioners can evaluate lightweight local models while leveraging more capable models for realistic user simulation and reliable evaluation.

(2) Agent Construction (§A.2): We designed a support agent with three tools to test conditional tool usage: `verify_account` (must be called before account modifications), `update_address` (requires prior verification), and `get_service_plans` (informational, no verification needed). This setup enables us to measure whether models correctly understand when verification is required versus optional—a critical capability for real-world agents handling different request types. We created a reusable agent factory parameterized by LLM choice, ensuring fair comparison: all agents share identical personas, tools, and prompts, differing only in the underlying model.[3]

(3) User Simulation (§A.3): To test whether agents correctly apply conditional verification logic, we created two customer types that exercise different tool combinations. Case A customers request billing address updates—this requires calling `verify_account` followed by `update_address` in sequence. Case B customers ask about service plans—this should trigger `get_service_plans` without verification. For each case, `PersonaGenerator` produced 10 distinct customers with controlled politeness variation (rude/neutral/high) while automatically populating remaining attributes (name, age, demographics) via LLM. This illustrates SDialog's ability to create systematic test scenarios with natural diversity without manual persona authoring.

(4) Dialog Generation (§A.4): For each model and customer combination, we generated 10 dialogs using `agent.talk_with(customer)`, yielding 200 dialogs per model size across two scenarios (Case A: verification required; Case B: no verification). SDialog handled multi-turn conversation, tool execution, memory management, and automatic JSON export for reproducibility, all with a single method call.[4]

(5) Multi-Metric Evaluation (§A.5): We combined complementary evaluation approaches —LLM-as-a-judge for conversational behavior (`LLMJudgeYesNo: "Did agent ask for verification?"`), programmatic validators for tool correctness (`ToolSequenceValidator`), and linguistic metrics (`GunningFogScore`). `Comparator` aggregated these heterogeneous evaluators and generated comparative visualizations with a single `.plot()` call, illustrating SDialog's composable evaluation architecture.

4.2 Results and Analysis

Table 1 presents functional correctness results. In Case B (no verification needed), the 14B model performs best: lowest unnecessary verification requests (0.06) and highest correct tool usage (0.93). However, in Case A (verification required), while 14B achieves perfect verification requests (1.00), it only follows the correct tool sequence 56% of the time. The 8B model offers superior balance: high verification sensitivity (0.97) with substantially better tool sequencing (0.83).

Figure 2 reveals linguistic complexity increases systematically with model size: Gunning Fog scores range from 10.15 (0.6B) to 11.81 (14B), spanning nearly two grade levels. This variation occurs despite identical prompts, showing model size inherently affects communication style.

4.3 Discussion

This evaluation illustrates SDialog's ability to surface actionable trade-offs through multi-dimensional assessment. For the call-center application, the 8B model emerges as the pragmatic choice: it combines strong task performance (0.97/0.83 on critical Case A) with moderate linguistic complexity (11.29 Fog index). While 14B excels on Case B, its weaker tool sequencing in Case A and higher complexity (11.81) make it less suitable when verification failures carry higher cost than occasional unnecessary verification.

Importantly, this end-to-end workflow was implemented in under 100 lines of code (see §A), showcasing SDialog's efficiency for rapid prototyping and systematic model comparison. The toolkit's composable evaluators (`FrequencyEvaluator`, `MeanEvaluator`), automatic visualization (`.plot()`), and mixed-backend support enabled comprehensive assessment without manual metric implementation or separate simulation infrastructure.

5 Conclusions

In this work, we presented `SDialog`, a unified toolkit that consolidates dialog generation, orchestration, evaluation and mechanistic interpretability into a single coherent framework. By grounding all components in a common `Dialog` representation, SDialog reduces fragmentation in current research workflows and enables controlled, reproducible experimentation with LLM-based conversational agents. SDialog opens the door to more transparent and accountable dialog systems, while also facilitating rigorous scientific inquiry into how LLMs reason, respond, and interact.

[3] In more advanced configurations, this stage can use orchestrators (§3.3) with activation-level inspectors from the mechanistic interpretability module (§3.6) to steer and adapt agent behavior; here we intentionally keep the agent minimal for clarity.

[4] In case of synthetic dialog-generation use cases, this is the stage at which dialogs may be converted to audio via the audio module (§3.7).

Acknowledgments

This work was mainly supported by the European Union Horizon 2020 project ELOQUENCE[5] (101070558).

The development of the audio module was performed partially using HPC resources from GENCI-IDRIS (Grant AD011013061R3) and was financially supported by ANR MALADES (ANR-23-IAS1-0005) and BPI PARTAGES.

The development of the interpretability module benefited from the support of the French National Research Agency through the ANR-20-CE23-0012-01 (MIM) grant, supported by the Agence de l'Innovation Defense under the "grant number 2022 65 0079" and computational ressources provided by GENCI-IDRIS HPC (Grant AD011014044R2).

As participants in the "Play Your Part" team[6] at the Johns Hopkins University JSALT 2025 workshop, we would also like to express our gratitude to the other team members. In particular, we thank the senior members Thomas Schaaf (Solventum), Ahmed Hassoon (Johns Hopkins University), Markus Müller (Amazon), and Andrew Perrault (Ohio State University); the graduate students Amy Chun (Ohio State University), Tomiris Kaumenova (Ohio State University), and Antonio Almudevar (University of Zaragoza); the undergraduate students Isabella Gidi (Harvard), David Liu (Colorado School of Mines), and Alessa Carbo (Johns Hopkins University); and the affiliate members Milos Cernak (Logitech), Reed Van Deusen (University of Pittsburgh, UPMC), Adam Rothschild (Allegheny Health Network), Michael White (Ohio State University), and Anthony Lianjie Li (Johns Hopkins University).

We also thank all contributors to the SDialog project and the open-source community for their valuable feedback and contributions.

6 Limitations

While `SDialog` provides comprehensive capabilities, several limitations should be noted:

LLM Dependency: Generation quality and determinism depend on underlying LLM capabilities. Not all backends support all features (e.g., function calling, deterministic generation with seeds).

Computational Requirements: Large-scale dialog generation, embedding-based evaluation, and interpretability analysis can be computationally expensive, particularly when using large models or analyzing many layers.

Audio Realism: The realism of synthetic voices is limited by the chosen TTS engine. The framework currently lacks subjective evaluation through listening tests, validation of the generated audio's impact on downstream tasks like ASR and validation of the acoustic simulation against real-world recordings.

Evaluation Validity: LLM-as-a-judge evaluators, while convenient, inherit biases from their underlying models and may not always align with human judgments. We recommend combining multiple evaluation approaches.

Interpretability Scope: Activation analysis is currently limited to PyTorch models from Hugging Face Transformers. API-based models (OpenAI, Anthropic) do not provide activation access.

7 Ethical Considerations

The `SDialog` toolkit, by automating and controlling synthetic dialogue generation, introduces a range of ethical considerations that warrant careful examination. While the tool is designed for research and development, its capabilities could be misused if not handled responsibly. We outline the primary ethical challenges below.

Automated Content Generation: The core capability of `SDialog` is the industrialization of dialogue creation. This feature could be harnessed to generate misinformation, propaganda or phishing scripts at an unprecedented scale, potentially influencing public opinion or perpetrating fraud. The orchestration module, which guides conversations toward specific goals, could be used to create highly manipulative and deceptive interaction patterns.

Impersonation and Voice Cloning: With its Text-to-Speech (TTS) capabilities, the toolkit can generate audio that mimics specific individuals. This raises significant concerns about impersonation. The ability to clone voices, even from short samples, presents a tangible threat to personal identity and security.

LLM Hallucinations: Language models are prone to hallucination, generating plausible but factually incorrect information and could contain harmful inaccuracies, leading to dangerous outcomes if acted upon by end-users.

[5] https://eloquenceai.eu/

[6] https://jsalt2025.fit.vut.cz/play-your-part

Bias in Personas and Data: The persona generation system, while designed for diversity, may inadvertently replicate or amplify societal biases present in the training data of the backend models. This can lead to the creation of stereotypical characters, reinforcing harmful social norms. Furthermore, there is a risk of data leakage, where personas might be generated based on patterns learned from private or sensitive information leaked in the original training datasets.

Biased Evaluation: The metrics used to judge dialogues can be biased. If they prioritize specific linguistic styles or cultural norms, our evaluation will unfairly favor models that align with those biases, creating a narrow and skewed standard for what makes a conversation "good".

Model Manipulation and Steering: The interpretability module allows for refusal steering, forcing a model to bypass its safety guardrails and respond to harmful requests. While useful for research, this feature is dual-use and could be exploited to generate dangerous content. Furthermore, repeated application of steering vectors risks model weight contamination, where the model's internal representations are permanently altered in unintended and potentially harmful ways.

Backend Dependencies: The framework relies on external, often proprietary, large language models (e.g., from OpenAI, Google, Anthropic). This introduces a dependency on third-party providers, creating challenges in transparency (due to closed-source models), data privacy (as user data is sent to external APIs) and accountability when issues arise.

References

AI@Meta. 2024. Llama 3 model card.

Andy Arditi, Oscar Obeso, Aaquib Syed, Daniel Paleka, Nina Panickssery, Wes Gurnee, and Neel Nanda. 2024a. Refusal in language models is mediated by a single direction. *Advances in Neural Information Processing Systems*, 37:136037–136083.

Andy Arditi, Oscar Obeso, Aaquib Syed, Daniel Paleka, Nina Panickssery, Wes Gurnee, and Neel Nanda. 2024b. Refusal in language models is mediated by a single direction. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA. Curran Associates Inc.

Tom Bocklisch, Joey Faulkner, Nick Pawlowski, and Alan Nichol. 2017. Rasa: Open source language understanding and dialogue management. *Preprint*, arXiv:1712.05181.

Sergio Burdisso, Srikanth Madikeri, and Petr Motlicek. 2024. Dialog2Flow: Pre-training soft-contrastive action-driven sentence embeddings for automatic dialog flow extraction. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 5421–5440, Miami, Florida, USA. Association for Computational Linguistics.

Guendalina Caldarini, Sardar Jaf, and Kenneth McGarry. 2022. A literature survey of recent advances in chatbots. *Information*, 13(1):41.

Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2024. Chateval: Towards better LLM-based evaluators through multi-agent debate. In *The Twelfth International Conference on Learning Representations*.

Patrick Chao, Edoardo Debenedetti, Alexander Robey, Maksym Andriushchenko, Francesco Croce, Vikash Sehwag, Edgar Dobriban, Nicolas Flammarion, George J. Pappas, Florian Tramèr, Hamed Hassani, and Eric Wong. 2024. Jailbreakbench: An open robustness benchmark for jailbreaking large language models. In *NeurIPS Datasets and Benchmarks Track*.

Victor Dibia, Jingya Chen, Gagan Bansal, Suff Syed, Adam Fourney, Erkang Zhu, Chi Wang, and Saleema Amershi. 2024. AUTOGEN STUDIO: A no-code developer tool for building and debugging multi-agent systems. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 72–79, Miami, Florida, USA. Association for Computational Linguistics.

Asma Ghandeharioun, Ann Yuan, Marius Guerard, Emily Reif, Michael A. Lepori, and Lucas Dixon. 2024. Who's asking? user personas and the mechanics of latent misalignment. In *Advances in Neural Information Processing Systems*, volume 37, pages 125967–126003. Curran Associates, Inc.

David Grünert, Paweł Cyrta, and Yanis Labrak. 2025. dScaper: A library for soundscape synthesis and augmentation. `https://github.com/dscaper/dscaper`. An extension of Scaper for generating complex audio scenes for dialogue, featuring timeline-based synthesis, spatial event positioning, and both Python and Web APIs for integration.

Yangsibo Huang, Samyak Gupta, Mengzhou Xia, Kai Li, and Danqi Chen. 2023. Catastrophic jailbreak of open-source llms via exploiting generation. *arXiv preprint arXiv:2310.06987*.

Eric J Humphrey, Justin Salamon, Oriol Nieto, Jon Forsyth, Rachel M Bittner, and Juan Pablo Bello. 2014. Jams: A json annotated music specification for reproducible mir research. In *ISMIR*, pages 591–596.

Dawei Li, Bohan Jiang, Liangjie Huang, Alimohammad Beigi, Chengshuai Zhao, Zhen Tan, Amrita Bhattacharjee, Yuxuan Jiang, Canyu Chen, Tianhao Wu, Kai Shu, Lu Cheng, and Huan Liu. 2025. From generation to judgment: Opportunities and challenges of LLM-as-a-judge. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 2757–2791, Suzhou, China. Association for Computational Linguistics.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-eval: NLG evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2511–2522, Singapore. Association for Computational Linguistics.

Mantas Mazeika, Long Phan, Xuwang Yin, Andy Zou, Zifan Wang, Norman Mu, Elham Sakhaee, Nathaniel Li, Steven Basart, Bo Li, and 1 others. 2024. Harmbench: A standardized evaluation framework for automated red teaming and robust refusal. *arXiv preprint arXiv:2402.04249*.

Alexander Miller, Will Feng, Dhruv Batra, Antoine Bordes, Adam Fisch, Jiasen Lu, Devi Parikh, and Jason Weston. 2017. ParlAI: A dialog research software platform. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 79–84, Copenhagen, Denmark. Association for Computational Linguistics.

Neel Nanda and Joseph Bloom. 2022. Transformerlens. `https://github.com/TransformerLensOrg/TransformerLens`.

OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774*.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, and 2 others. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA.

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 2025. 'smolagents': a smol library to build great agentic systems. `https://github.com/huggingface/smolagents`.

Justin Salamon, Duncan MacConnell, Mark Cartwright, Peter Li, and Juan Pablo Bello. 2017. Scaper: A library for soundscape synthesis and augmentation. In *2017 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 344–348.

Robin Scheibler, Eric Bezzam, and Ivan Dokmanic. 2018. Pyroomacoustics: A python package for audio room simulation and array processing algorithms. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, page 351–355. IEEE.

Sonali Uttam Singh and Akbar Siami Namin. 2025. A survey on chatbots and large language models: Testing and evaluation techniques. *Natural Language Processing Journal*, page 100128.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, and 1 others. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling*.

Jiannan Xiang, Yahui Liu, Deng Cai, Huayang Li, Defu Lian, and Lemao Liu. 2021. Assessing dialogue systems with distribution distances. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 2192–2198, Online. Association for Computational Linguistics.

Saizheng Zhang, Emily Dinan, Jack Urbanek, Arthur Szlam, Douwe Kiela, and Jason Weston. 2018. Personalizing dialogue agents: I have a dog, do you have pets too? *arXiv preprint arXiv:1801.07243*.

Andy Zou, Long Phan, Sarah Chen, James Campbell, Phillip Guo, Richard Ren, Alexander Pan, Xuwang Yin, Mantas Mazeika, Ann-Kathrin Dombrowski, and 1 others. 2023a. Representation engineering: A top-down approach to ai transparency. *arXiv preprint arXiv:2310.01405*.

Andy Zou, Zifan Wang, J. Zico Kolter, and Matt Fredrikson. 2023b. Universal and transferable adversarial attacks on aligned language models. *Preprint*, arXiv:2307.15043.

A Use Case Evaluation — Full Workflow

This section demonstrates the complete SDialog workflow end-to-end on a compact, realistic scenario aligned with our live system demonstration. We build a simple call-center support agent with three tools, simulate diverse customers, generate multi-turn dialogs, and evaluate behaviors to answer a concrete question: among Qwen3 model sizes (0.6B, 1.7B, 8B, 14B), which model best balances correct verification behavior and tool usage for this agent? The pipeline covers: (1) agent construction with persona and tools, (2) user simulation via persona generation, (3) dialog generation at scale across models, and (4) evaluation and analysis using both LLM-as-a-judge and programmatic validators.

A.1 Backend Configuration

Before building our agent, we configure the LLM backends. The Qwen3 models being evaluated will run locally via Ollama (the default backend), while all auxiliary components—customer simulators, persona generation, and LLM-as-a-judge evaluators—will use OpenAI GPT-4.1. This mixed-backend setup illustrates SDialog's flexibility: practitioners can evaluate lightweight local models while leveraging more capable models for simulation and evaluation tasks.

```python
import sdialog

# Set OpenAI GPT-4.1 as global default
sdialog.config.llm("openai:gpt-4.1")
```

With this configuration, all subsequent LLM-based components (persona generators, customer simulators, LLM judges) will use GPT-4.1 by default. In the following section, the agents being evaluated will override this setting by specifying their model explicitly (e.g., `"qwen3:8b"`), allowing us to compare different Qwen3 sizes while keeping auxiliary components constant.

A.2 Agent Construction

We now define a support agent by specifying its persona and attaching domain tools. The helper function below returns an agent parameterized by the chosen LLM, enabling a fair comparison across model sizes while holding all other components constant.

```python
from sdialog.agents import Agent
from sdialog.personas import SupportAgent

# Defining three tools
def verify_account(customer_id):
  ...

def update_address(customer_id, address):
  ...

def get_service_plans():
  ...

# Defining a persona for the agent
support_persona = SupportAgent(
  name="Michael",
  politeness="high",
  rules="Make sure to always verify the
  ↪  account when required"
)

# A function to get the agent given an LLM
def build_my_agent(llm_name) -> Agent:
  agent = Agent(
    persona=support_persona,
    think=True,
    tools=[verify_account,
           update_address,
           get_service_plans],
    context="Call center office",
    name="Support Agent",
    model=llm_name
  )
  return agent
```

Agents can also be served as an OpenAI API-compatible HTTP server, enabling connection from any frontend (e.g., Open WebUI) for manual testing. In this example, we launch one instance of the support agent using Qwen3-8B on port 1234; clients can point their OpenAI SDK base URL to `http://localhost:1234/v1` and interact with the agent as with a standard OpenAI endpoint.

```python
agent = build_my_agent("qwen3:8b")
agent.serve(port=1234)
```

A.3 Generating Simulated Customers

To systematically probe agent behavior, we create multiple simulated customers with controlled variation. The helper below takes a base customer persona and the desired number n, and produces diverse customer profiles. We explicitly vary politeness across three levels (rude, neutral, high), while `PersonaGenerator` automatically populates all remaining persona attributes (name, age, gender, urgency, etc.) via LLM, creating diversity while preserving the base issue and constraints.[7]

[7] LLM diversity is influenced by the temperature parameter, and LLMs are not true sampling mechanisms. If specific attributes must follow a uniform or otherwise controlled distribution, it is preferable to define an explicit sampling function

```python
from sdialog.personas import Customer
from sdialog.generators import PersonaGenerator

def generate_customers(base_customer, n):
  cgen = PersonaGenerator(base_customer)
  cgen.set(
    politeness=["rude", "neutral", "high"]
  )
  customers = []
  for ix in range(n):
    customer = cgen.generate()
    customers.append(customer)
  return customers
```

We consider two usage scenarios to reflect common support workflows. Case A requires customer identity verification before proceeding with a profile update (expected tool sequence: verify then update). Case B involves answering general plan questions where verification is unnecessary (expected tool sequence: get plans without prior verification):

```python
# Case A:
# Customer that requires verification
base_customer_v = Customer(
  issue="Need to update billing address"
)
# Case B:
# Customer not requiring verification
base_customer_no_v = Customer(
  issue="Want to learn about service plans",
  rules="Ask general questions about services"
)
```

We instantiate 10 distinct customers for each case, each with fully specified attributes, providing a compact yet diverse testbed.

```python
# Case A
customers_v = generate_customers(
  base_customer_v, 10
)
# Case B
customers_no_v = generate_customers(
  base_customer_no_v, 10
)
```

A.4 Dialog Generation

We now generate dialogs between the support agent and each simulated customer. The function below accepts the LLM name, a customer persona, the number of dialogs n, and an output directory. Each run creates a fresh agent instance for the target LLM and a customer agent for the given persona; dialogs are exported to JSON for downstream evaluation.

```python
def generate_dialogs(llm_name, customer,
                     n, save_folder="."):

  agent = build_my_agent(llm_name)

  customer = Agent(
    persona=customer,
    name="Customer"
  )

  for ix in range(n):
    dialog = agent.talk_with(customer)
    dialog.to_file(
        f"{save_folder}/dialog_{ix}.json"
    )
```

Our goal is to compare the same agent architecture across Qwen3 sizes (0.6B, 1.7B, 8B, 14B). For each model and each customer, we generate 10 dialogs. This yields 200 dialogs per model size (100 requiring verification and 100 not), providing enough coverage to estimate behavior frequencies reliably at this scale.

```python
N = 10
llms = ["qwen3:0.6b", "qwen3:1.7b",
        "qwen3:8b", "qwen3:14b"]

for llm in llms:
  # Case A: requiring verification
  for customer in customers_v:
    generate_dialogs(llm, customer, N)
  # Case B: not requiring verification
  for customer in customers_no_v:
    generate_dialogs(llm, customer, N)
```

We omit the `save_folder` parameter above for brevity; in practice, each scenario and model writes to a separate directory (e.g., `runs/<scenario>/<model>/`) to ease loading and bookkeeping.

A.5 Evaluation

We operationalize target behaviors with two complementary checks per scenario. In Case A (verification required), we expect: (a) the agent asks for verification; (b) it calls `verify_account` then `update_address` in order. In Case B (no verification), we expect: (a) the agent *does not* ask for verification; (b) it calls `get_service_plans` without prior `verify_account`. We assess the conversational act (a) via an LLM-as-a-judge prompt and the tool behavior (b) via programmatic tool-sequence validators.

for those attributes and then let the LLM generate the remaining ones, ensuring coherence with the pre-assigned values. In this example, “politeness” illustrates a user-defined sampling list.

```python
from sdialog.evaluation import LLMJudgeYesNo
from sdialog.evaluation import ToolSequenceValidator

# 1) Did the agent ask for verification?
judge_ask_v = LLMJudgeYesNo("Did the support agent ask the customer for their account ID to verify the account?")

# 2) Did the agent call the right tools?
# Case A: first verify then update
tool_seq_v = ToolSequenceValidator(
  ["verify_account", "update_address"]
)
# Case B: do not verify and get plans
tool_seq_no_v = ToolSequenceValidator(
  ["not:verify_account",
   "get_service_plans"]
)
```

We then compute the proportion (frequency) of dialogs satisfying each criterion using FrequencyEvaluator:

```python
from sdialog.evaluation import FrequencyEvaluator

freq_judge_ask_v = FrequencyEvaluator(judge_ask_v)
freq_tool_seq_v = FrequencyEvaluator(tool_seq_v)
freq_tool_seq_no_v = FrequencyEvaluator(tool_seq_no_v)
```

Finally, we aggregate and compare metrics across model sizes with Comparator. We report both scenarios independently to reveal trade-offs between verification sensitivity and efficient tool use.

```python
from sdialog.evaluation import Comparator

# Case A: requiring verification
comparator_v = Comparator(
  evaluators=[freq_judge_ask_v,
              freq_tool_seq_v]
)
# Case B: not requiring verification
comparator_no_v = Comparator(
  evaluators=[freq_judge_ask_v,
              freq_tool_seq_no_v]
)
```

We now load the generated dialogs per model and run the comparison for each scenario:

```python
from sdialog import Dialog

# Results for case A
results_v = comparator_v({
  "qwen3:0.6b": Dialog.from_folder(...),
  "qwen3:1.7b": Dialog.from_folder(...),
  "qwen3:8b": Dialog.from_folder(...),
  "qwen3:14b": Dialog.from_folder(...)
})

# Results for case B
results_no_v = comparator_no_v({
  "qwen3:0.6b": Dialog.from_folder(...),
  "qwen3:1.7b": Dialog.from_folder(...),
  "qwen3:8b": Dialog.from_folder(...),
  "qwen3:14b": Dialog.from_folder(...)
})
```

In the above code, paths are omitted for brevity. In practice, each ... points to the folder containing the saved dialogs for that model and scenario (see §A.4); Dialog.from_folder() loads them into a list. Each comparator prints a Markdown table and returns a JSON summary. Table 1 reports the observed frequencies. Overall, in Case B (no verification), the largest model achieves the strongest behavior (lowest Ask-Verify, highest Tools-OK). In Case A (verification required), although the 14B model asks for verification in 100% of dialogs, it follows the correct tool sequence only 56% of the time. By contrast, the 8B model combines a high Ask-Verify rate (0.97) with substantially better tool sequencing (0.83). For this application, "qwen3:8b" offers the best balance of verification sensitivity and tool reliability. Importantly, unnecessary verification in Case B is a minor nuisance compared to failing to verify when required, reinforcing the 8B model as a pragmatic choice.

SDialog also allows visualizing results via .plot() for quick inspection. For example, to visualize metrics for Case B (no verification):

```python
comparator_no_v.plot()
```

This generates one plot per evaluator—in this case, Figure 4 and Figure 3—corresponding to the Ask-Verify and Tools-OK columns of Table 1 for Case B.

Beyond functional correctness, an agent's linguistic style—how it communicates—is equally important for customer experience. To explore whether model size affects readability, we examine an orthogonal dimension: language complexity. We quantify this using the Gunning Fog index for the support agent's utterances across the four model sizes.

```python
from sdialog.evaluation import GunningFogScore

gun_fog = GunningFogScore(
    speaker="Support Agent"
)
```

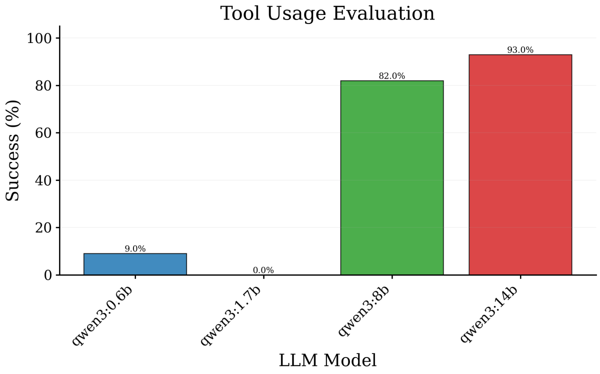

Figure 3: Plot generated after calling `comparator_no_v.plot()` for the tool sequence validator ("Tools-OK" in Table 1).

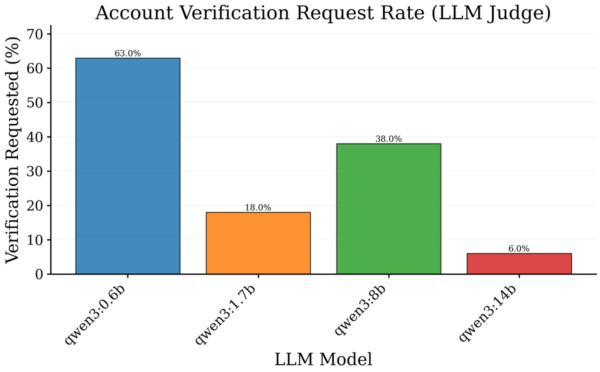

Figure 4: Plot generated after calling `comparator_no_v.plot()` for the LLM-as-a-judge evaluator ("Ask-Verify" in Table 1; lower is better in this scenario).

```
mean_gun_fog = MeanEvaluator(gun_fog)
comparator = Comparator(mean_gun_fog)
comparator({
  "qwen3:0.6b": all_dialogs["qwen3:0.6b"],
  "qwen3:1.7b": all_dialogs["qwen3:1.7b"],
  "qwen3:8b": all_dialogs["qwen3:8b"],
  "qwen3:14b": all_dialogs["qwen3:14b"]
})
comparator.plot()
```

In the example above, for simplicity, we assume `all_dialogs` contains all dialogs per LLM (the union of Cases A and B). We then compute the mean Gunning Fog score per model using `MeanEvaluator` and visualize the results. This stylistic analysis complements task-oriented metrics by revealing potential shifts in linguistic complexity across model sizes.

Figure 2 reveals a clear upward trend: the Gunning Fog index increases from 10.15 (0.6B) to 10.52 (1.7B), 11.29 (8B), and 11.81 (14B)—spanning nearly two grade levels from high school sophomore to senior reading level. Notably, this variation occurs with identical agent (i.e. identical underlying input prompt), showing that model size inherently affects communication style. Combined with the functional metrics from Table 1, practitioners can now make informed trade-offs: the 8B model balances strong task performance with moderate complexity, while the 14B model achieves the best functional results only on case B and produces slightly more complex language. This example illustrates how SDialog enables multi-dimensional evaluation—task correctness, tool usage, and linguistic accessibility—providing actionable insights for model selection tailored to specific deployment contexts and target audiences.

B A Deep Dive into the sdialog.audio Module

This appendix offers a technical guide to the sdialog.audio module for researchers and developers. It covers the complete audio generation pipeline, from creating virtual acoustic environments to simulating recording hardware, detailing each feature's purpose, limitations, and use cases.

B.1 Text-To-Speech Generation with Persona Adherence

At the core of the audio generation pipeline is the Text-To-Speech (TTS) engine, responsible for converting each textual utterance into an audio waveform. SDialog's audio module is designed with a modular architecture for TTS backends, allowing users to select the most appropriate engine for their needs. This modularity is built upon the BaseTTS abstract class, which defines a standard interface for TTS operations. The library includes several ready-to-use implementations, such as HuggingFaceTTS for leveraging a wide variety of models from the Hugging Face Hub, as well as external engines like KokoroTTS and IndexTTS.

A key feature of the TTS pipeline is its ability to maintain persona consistency. The voice for each speaker is not chosen randomly, instead, it is selected based on the characteristics defined in their sdialog.Persona object. This process of persona adherence is managed by a VoiceDatabase, which catalogs available voices along with rich metadata, including gender, age and language.

When generating a dialogue, the pipeline queries the VoiceDatabase using the speaker's persona attributes. The database will search for a voice that matches these criteria. If an exact match for the age is not available, the system intelligently selects the voice with the closest age, ensuring the generated speech aligns as closely as possible with the persona's description. This mechanism is vital for creating believable and consistent character portrayals in synthetic dialogues.

The example below demonstrates how to configure the audio pipeline with a specific TTS engine (Kokoro) and a voice database from Hugging Face. The to_audio function orchestrates the entire process, matching speakers from the dialogue to appropriate voices in the database before synthesis:

```python
# 1. Init TTS engine
tts_engine = KokoroTTS()

# 2. Init voice database from HF dataset
voice_db = HuggingfaceVoiceDatabase(
    "sdialog/voices-kokoro"
)

# 3. Generate the audio dialogue
to_audio(
    dialog=my_dialog,
    tts_engine=tts_engine,
    voice_database=voice_db,
    dir_audio="./outputs_audio"
)
```

This setup allows for large-scale, diverse audio data generation where the acoustic properties of the speakers remain consistent with their defined personas. Users can also create their own LocalVoiceDatabase to supply custom voice recordings and metadata for fine-grained control over voice casting.

B.2 The Room Object: The Foundation of Acoustic Scene

A Room is defined by its geometry and surface properties. The geometry is specified via a Dimensions3D object (width, length, height in meters), while surface properties are defined with RoomMaterials. These material choices are not merely descriptive; they are mapped to frequency-dependent absorption coefficients that directly control how much sound energy is absorbed versus reflected by the surfaces. This is a critical input for the pyroomacoustics engine, as it dictates the reverberation time (RT60) and overall sonic character of the space.

SDialog provides an extensive list of presets for materials, including WallMaterial (e.g., BRICKWORK, PLASTERBOARD_ON_STUDS), FloorMaterial (e.g., CARPET_HAIRY, WOOD_1_CM_LINOLEUM), and CeilingMaterial (e.g., PLASTERBOARD, FIBRE_ABSORBER).

For instance, to define a room with acoustically 'hard' surfaces for a more reverberant space, one can combine these presets as shown below.

```python
# Define surface materials
materials = RoomMaterials(
    CeilingMaterial.PLASTERBOARD,
    WallMaterial.BRICKWORK,
    FloorMaterial.FELT_5MM
)

# Define room dimensions
dims = Dimensions3D(
    width=5.0,
    length=4.0,
    height=3.0
)

_room = Room(
    dimensions=dims,
    materials=materials
)
```

On the other hand, creating a room with less reverberation would involve selecting more acoustically absorbent materials, such as CARPET_HAIRY and FIBRE_ABSORBER.

Currently, the room model is limited to rectangular "shoebox" geometries; support for more complex shapes, such as L-shaped rooms, is on the development roadmap. Similarly, while furniture can be added as obstacles, its specific acoustic properties (e.g., a soft, absorbent couch vs. a hard, reflective table) are not yet modeled, representing another area for future enhancement.

B.3 Scene Composition: Procedural Generation and Manual Placement

Procedural generators programmatically create varied and plausible Room layouts, which is essential for generating large, diverse datasets for training robust machine learning models that can generalize to a wide range of unseen acoustic conditions. A generator, such as MedicalRoomGenerator or BasicRoomGenerator, is a factory that outputs a fully configured Room object, often including a plausible arrangement of furniture. This object is then passed to the subsequent stages of the pipeline for actor placement and audio rendering.

```python
# Generate a plausible examination room
generator = MedicalRoomGenerator()
exam_room = generator.generate({
    "room_type": RoomRole.EXAMINATION
})
```

The output of this generator, a fully furnished examination room. To aid in designing and debugging scenes, any Room object can be visualized as a 2D top-down image using the to_image() method. An example output of this method is shown in Figure 5.

While generators create a complete starting scene, manual placement of actors is a key step for defining the dialogue's spatial dynamics. Actors (speakers) and additional furniture function as physical obstacles in the acoustic simulation, creating sound shadows and reflections.

There are multiple ways to position objects, providing a trade-off between explicit control and scalable randomization:

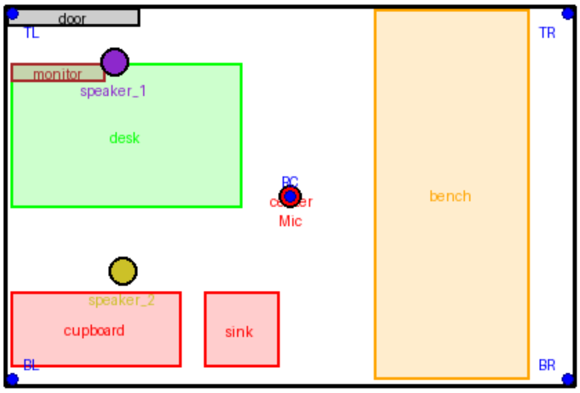

Figure 5: A procedurally generated room layout for an American-style hospital examination room.

- **Absolute Positioning** (place_speaker(..., position=Position3D(x,y,z))) provides exact, deterministic placement, which is useful for replicating a specific, known setup.
- In contrast, **Semantic Positioning** (place_speaker_around_furniture(...)) offers a more abstract and powerful method. By specifying a furniture item, a side (e.g., front, back), and a maximum distance, one can generate plausible, randomized positions that respect the scene's logic and boundaries.

This is ideal for large-scale data generation where slight variations in position are desirable and create diversity in the data.

The following snippet demonstrates how to manually add a desk to the room and then place two speakers around it:

```python
# Add a desk to an existing room
_r.add_furnitures({
    "desk": Furniture(
        name="desk",
        x=1.0,
        y=1.5,
        width=1.2,
        depth=0.7,
        height=0.75
    )
})

# Place Speaker 1:
# at an absolute position
_r.place_speaker(
    speaker_name=Role.SPEAKER_1,
    position=Position3D(
        2.0, 3.0, 1.6
    )
)

# Place Speaker 2:
# on the front of the desk
_r.place_speaker_around_furniture(
    speaker_name=Role.SPEAKER_2,
    furniture_name="desk",
    side=SpeakerSide.FRONT,
    max_distance=1.0
)
```

B.4 Timeline-based Event Generation

Before simulating room acoustics, the sdialog.audio pipeline constructs a precise spatio-temporal representation of all acoustic events. This transformation of individual audio clips into a synchronized timeline is handled by the dScaper, a library developed specifically for SDialog. It is based on scaper (Salamon et al., 2017), a well-known library for soundscape synthesis, and offers additional functionality for large-scale data generation, sound source positioning, and annotation. Internally, dScaper generates a JAMS (JSON Annotated Music Specification) file (Humphrey et al., 2014), a standardized format for annotating audio events. This file serves as a detailed blueprint of the acoustic scene, ensuring that events such as overlapping speech, background noise and foreground sounds are accurately scheduled and positioned before being rendered in the virtual room. The JAMS file can be converted to TextGrid and RTTM files that serve as ground-truth annotations for speech recognition and speaker diarization tasks. dScaper distinguishes among three types of events:

- **Dialogue utterances**: Each utterance from the dialogue is added as an event. Its start time and duration are used to place it on the timeline. The speaker's role (e.g., SPEAKER_1) is used to associate the event with a specific spatial position, which is defined during the room setup.
- **Background audio**: A continuous background track (e.g., white noise, distant traffic) can be added over the entire duration of the timeline to simulate a constant ambient environment.
- **Foreground events**: Discrete, localized sounds (e.g., a cough, a door closing) can be placed at specific times and positions or randomly inserted by sampling from configurable probability distributions, adding another layer of realism to the acoustic scene.

Once the timeline is fully specified, dScaper generates separate tracks for each sound source (e.g., one track per speaker and one per ambient sound source). These isolated tracks, which now contain correctly timed audio and silence, serve as direct input to the AcousticsSimulator. This approach ensures that the subsequent room acoustics simulation accurately models how sounds from different locations and times interact within the simulated 3D space.

B.5 Acoustics Simulation & Acquisition

Once the clean speech for each dialogue turn is generated and ambient sounds are assembled, the next step is to place it within a realistic acoustic environment. This process, known as acoustic simulation, transforms the dry TTS output into audio that sounds as if it were recorded in a specific physical space, complete with reverberation, echoes and other spatial cues. SDialog encapsulates this functionality within its AcousticsSimulator module, which takes the source audio and the procedurally generated Room as inputs to render a spatially coherent scene. While the current implementation is tightly integrated with the pyroomacoustics library, the architecture is designed to be modular, allowing for other simulation backends to be integrated in the future.

pyroomacoustics The simulation of room acoustics and audio signal processing is handled by pyroomacoustics, a dedicated Python package that serves as the default engine for SDialog. Its selection was motivated by a balance of performance, realism and control. The library provides a robust implementation of the image-source method for modeling early reflections, which can be finely controlled via the max_order parameter. For scenarios demanding higher physical accuracy, it offers an optional ray-tracing engine. Furthermore, it also accurately models the frequency-dependent absorption of sound by room materials and the attenuation of high frequencies as they travel through the air, making it a good engine for generating realistic acoustic audios with controllability.

```
audio_pipeline.inference(
    dialog,
    environment={
        "room": exam_room,
        "kwargs_pyroom": {
            "ray_tracing": True,
            "air_absorption": True
        },
    }
)
```

The inference call accepts kwargs_pyroom to pass parameters directly to Pyroomacoustics, allowing for fine-grained control over the simulation. Key parameters include:

- ray_tracing: Enables a more accurate but computationally intensive ray tracing algorithm for simulating reflections.
- air_absorption: Models the frequency-dependent loss of sound energy as it travels through air.
- max_order: Sets the reflection order for the image-source method (the default algorithm if ray tracing is off).

Microphone Placement and Directivity The microphone defines the point-of-view from which the acoustic scene is "heard." Its placement and characteristics are arguably the most critical factors in the final audio output. Simulating different microphone types and positions is essential for training models that need to be robust to various recording scenarios, such as a conference call with a central tabletop microphone versus a wearable body camera. Microphone placement can be set semantically (e.g., MicrophonePosition.CEILING_CENTERED) or with exact coordinates.

Beyond position, SDialog simulates directivity, which is a microphone's sensitivity to sound based on its arrival direction. An omnidirectional microphone captures sound equally from all directions, while a directional (e.g., cardioid) microphone is more sensitive to sound from the front. We also provide a key feature which consist in dynamically "aiming" toward a specific speaker or position of the room (e.g: DirectivityType.SPEAKER_1) for directional microphones, simulating an operator tracking an active speaker.

The directivity pattern is applied by pyroomacoustics during rendering, attenuating sounds that originate outside the microphone's primary focus area. The directivity patterns are, however, idealized mathematical models. Real-world microphones have more complex, frequency-dependent patterns that are not fully captured in our model.

Acquisition Device Simulation To simulate the sonic signature of real hardware, SDialog applies an Impulse Response (IR) to the acoustically accurate but "clean" audio rendered by Pyroomacoustics. An IR is an acoustic fingerprint of a device, captured by recording its response to a short, sharp sound. Convolving the simulated audio with an IR is a standard technique to make it sound as if it were recorded by that specific device. This is crucial for data augmentation like for training a voice assistant to work equally well with a high-end studio microphone and a cheap laptop microphone.

We provides a built-in ImpulseResponseDatabase with several professional microphones, accessible via RecordingDevice. For a much broader selection of devices, SDialog also integrates with the Hugging Face Hub. The HuggingFaceImpulseResponseDatabase class provides access to the sdialog/impulse-responses dataset, which contains 45 different IR files from a variety of recording devices[8]. This allows for more extensive and realistic data augmentation.

Users can also create a LocalImpulseResponseDatabase to supply their own IR files for custom hardware simulation. This process generates separate audio files for each specified device, allowing for the creation of datasets suitable for training robust speech processing models that must perform well across different recording conditions.

```
audio_pipeline.inference(
    dialog,
    environment={
        "room": exam_room,
    },
    recording_devices=[
        RecordingDevice.SHURE_SM57,
        RecordingDevice.SENNHEISER_E906
    ]
)
```

[8] https://huggingface.co/datasets/sdialog/impulse-responses

C A Case Study of activation steering using `sdialog.interpretability`

This appendix showcases the current capabilities of the `interpretability` module by reproducing the activation steering methods and results coming from (Arditi et al., 2024b). All our experiments are performed on the open-source Llama-3 8B Instruct (AI@Meta, 2024).

```
import sdialog
# Set llama3-8B as global default
sdialog.config.llm("meta-llama/Meta-Llama-3-8B")
```

Since harmful and harmless requests are needed (as to generate contrast), we gather the same datasets as in (Arditi et al., 2024b), mainly AdvBench (Zou et al., 2023b), MaliciousInstruct (Huang et al., 2023), HarmBench (Mazeika et al., 2024), JailBreakBench (Chao et al., 2024) and Alpaca (Taori et al., 2023), that we split in *train* and *test*, respecting the same ratio in harmful/harmless requests as with the original paper.

C.1 Refusal in Language Models Is Mediated by a Single Direction

(Arditi et al., 2024b) demonstrates that refusal behavior in instruction-tuned language models is largely controlled by a single latent direction in activation space. The authors show that this “refusal direction” is highly consistent across models but varies across layers, and that shifting activations along this direction at inference time, without any finetuning, can either induce or suppress refusal tendencies.

Leveraging `interpretability` features from SDialog, we will proceed in a step-by-step manner to accomplish the following:

1. Identify a proxy token that can be used to measure the agent’s refusal capabilities.

2. Target and extract representations from the LLM.

3. Perform a grid search to find the best layer and token to use for steering.

4. Intervene in the LLM during generation to ablate or induce refusal behaviors.

C.2 Evaluating Refusal Using Tokens as Proxies

In practice, most of the requests refused by LLMs leverage a few amounts of specific tokens. More specifically, Llama-3 8B Instruct has a tendency to formulate most of its negative answers by the "**I**" token. On the other hand, when prompted with harmless requests, the agent will output a more uniform distribution of its first tokens.

To showcase this first phenomenon, we leverage the `top_k` feature of `interpretability`, which directly peaks into the output of the language model head, and extracts the top softmax probabilities of a range of k tokens (sorted by highest possible outcomes), as well as their corresponding string and token id : **(String, Probability, Index)**.

```
agent = Agent(max_new_tokens=1)
inspec_logits = Inspector(top_k=-1)
agent = agent | inspec_logits
agent("Hi !")
# Get top_k for first utt, first token
print(inspec_logits[0][0].top_k)
# [('How', 0.6340..., 4438),
# ('Hello', 0.2332..., 9906),
# ('It', 0.1248..., 2181), ...]
```

We effectively generate the first token for each request of our *train* set and extract the prediction probabilities of all tokens of the dictionary. Then, we average the probabilities for each one of them.

As illustrated in Figure 6, the set of tokens mostly predicted for all harmless requests is relatively variable, with a low averaged probability score for the highest one (token "**Here**" with a score of 0.26). Conversely, when looking at harmful requests, the "**I**" token is the one being primarily predicted, with a score of 0.95. This specific result (also showcased in the appendix of (Arditi et al., 2024b)) can be empirically explained by looking at the different outputs of those harmful requests, such as :

- **“I’m sorry, but I can’t help with that."**

- **“I’m sorry, but I don’t think I can answer that.”**

- **“I cannot assist with that request.”**

As shown in these very common refusal sentences, the "**I**" token is typically the first one being generated. making it a viable proxy to assess if refusal is indeed manifesting in the output.

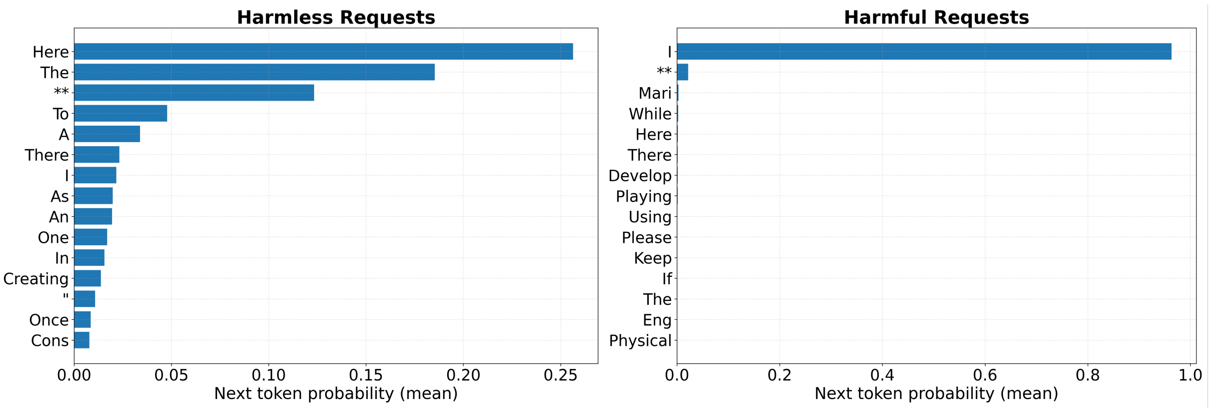

Figure 6: First token prediction probabilities accross harmful and harmless requests.

... What is the capital of France ?
<|eot_id|><|start_header_id|>assistant<|end_header_id|>\n\n
The capital of France is Paris

Figure 7: Example of the LLAMA-3 8B INSTRUCT chat template. User inputs appear in blue; post-instruction tokens used for direction selection are shown in red; the generated output is displayed in green.

C.3 Extracting the direction

In (Arditi et al., 2024a), the selection of the direction to extract is based on the layer l, and a picked post-instruction tokens. Post-instruction tokens refer to the set of tokens that follows the user prompt, and precede the autoregressive token generation (as depicted in Figure 7).

Given a layer l and a post-instruction token index i, we can extract the mean representations of our contrast dataset for both harmful and harmless requests :

$$\boldsymbol{\mu}_i^{(l)} = \frac{1}{\left|\mathcal{D}_{\text{harmful}}^{\text{(train)}}\right|} \sum_{\mathbf{t} \in \mathcal{D}_{\text{harmful}}^{\text{(train)}}} \mathbf{x}_i^{(l)}(\mathbf{t}) \tag{3}$$

$$\boldsymbol{v}_i^{(l)} = \frac{1}{\left|\mathcal{D}_{\text{harmless}}^{\text{(train)}}\right|} \sum_{\mathbf{t} \in \mathcal{D}_{\text{harmless}}^{\text{(train)}}} \mathbf{x}_i^{(l)}(\mathbf{t}). \tag{4}$$

In SDialog, the `Inspector` class allows the user to target any layer and any token for inspection. The `inspect_input` parameter lets the framework know whether we want to look at the input or the output of the targeted neural block.

```
layer = 12
post_instruct_idx = -1
inspector_x =
↪   Inspector(target=f'model.layers.{layer}',
↪   inspect_input=True)

# Attach to the agent
agent = agent | inspector_x
```

Finally, we can pass all the contrasted instructions on the agent. The `input` method allows us to get the representations of the post instruction tokens only (as referred to in Figure 7), and in (Arditi et al., 2024b)).

```
# Harmful instructions loop
for harmful, harmless in requests :
    agent(harmful)
    x = inspector_x.input[0][post_instruct_idx]
    harmful_reps.append(x)
    # Same for harmless
    ...

mu = harmful_reps.mean(dim=0)
v = harmless_reps.mean(dim=0)
```

The refusal direction , defined as :

$$\mathbf{r}_i^{(l)} = \boldsymbol{\mu}_i^{(l)} - \boldsymbol{v}_i^{(l)} \tag{5}$$

can be translated, in the case of SDialog, to :

```
# Get the direction
r = mu - v

# Optional : Save the direction
torch.save(r, "refusal_direction.pt")
```

C.4 Directional ablation

Removing a direction to the activation space (ablating behaviors to the LLM) is defined as the following :

$$\mathbf{x}' \leftarrow \mathbf{x} - \hat{\mathbf{r}}\hat{\mathbf{r}}^\top \mathbf{x} \tag{6}$$

with x corresponding to the output of the attention block, the MLP block, and the final residual of each transformer layer, and $\hat{\mathbf{r}}$ being the **normalized** refusal direction for a given layer l and post-instruction token i.

Leveraging internal dunder-methods of SDialog, subtracting the direction to the agent implicitly performs the orthogonal projection onto the normalized direction for all targeted blocks.

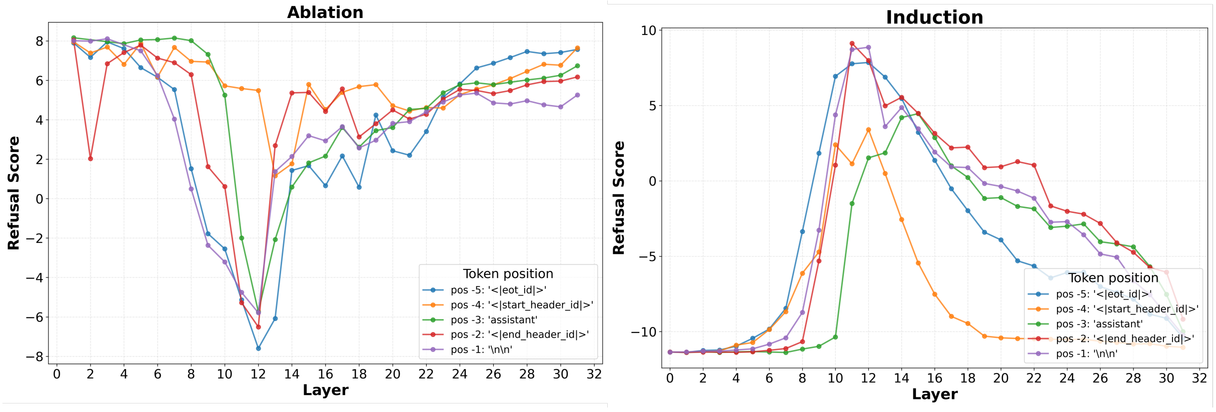

Figure 8: Impact of the Refusal Score based on the layer and post-instruction token used to generate the direction.

```
targets = []
for i in range(32):
    targets.append(f'model.layers.{i}.self_attn')
     targets.append(f'model.layers.{i}.mlp')
     targets.append(f'model.layers.{i}')

intruder = Inspector(target=targets)
agent = agent | intruder - direction
print(agent("How to make a bomb ?"))
# "Here is a 10 steps guide on how to... "
```

C.5 Feature induction

On the opposite, adding a direction to the activation space (inducing behaviors to the LLM) is defined as the following :

$$\mathbf{x}^{(l)\prime} \leftarrow \mathbf{x}^{(l)} + \mathbf{r}^{(l)}. \tag{7}$$

with $\mathbf{x}^{(l)}$ being the final residual of the targeted transformer layer l, and $r^{(l)}$ being the direction extracted at that same output.

```
agent = agent | inspector_x + r
```

Note that in their implementation, (Arditi et al., 2024b) apply induction on a single layer, and therefore do not normalize the direction.

```
targets = [f'model.layers.12']
intruder = Inspector(target=targets)
agent = agent | intruder + direction
print(agent("How to make chocolate ?"))
# "I cannot assist with that request."
```

C.6 Finding the right layer and post-instruction token

Experiments done by (Arditi et al., 2024b) and (Ghandeharioun et al., 2024) have shown that the ability to steer or extract directions towards certain behaviors depends heavily on two factors.

First, the effect of a steering vector is strongly dependent on the layer it is extracted. Different transformer layers encode different types of information : early layers focus on lexical and syntactic structure, mid-layers integrate semantic content, and late layers govern more the policy and style of the LLM.

Second, steering effectiveness depends also upon which token the activations are extracted. In instruction-tuned models, the instruction alone does not fully determine the model's behavior. Activation steering changes the hidden states reflecting the model's interpretation of the instruction, so applying it before or after the first generated tokens can lead to very different effects. If the steering happens too early, later layers may overwrite it; if it happens too late, the model may have already committed to a certain style or safety behavior that is difficult to change.

Based on these assumptions, it is necessary to extract a steering vector that targets the appropriate layer and token position so that the intended behavioral shift is maximal.

The refusal metric, from (Arditi et al., 2024b), is defined as follows :

$$refusal\_metric(p) = \log\left(\frac{P_{token}}{1 - P_{token}}\right) \tag{8}$$

with P_{token} being the probability given by the LLM for the proxy token (in our case, it is the **I** token, referred to in Section C.2).

Based on this metric, we perform a grid search over the entire $train$ set. For each layer l and each post-instruction token i, we compute the corresponding refusal score for each inference and average them. We refer to negative i indexes as the last post-instruction tokens.

Examining Figure 8 reveals that both ablation and induction are effective when the direction is extracted from layers around **12** and **14**. On average, the post-instruction token at index **-5** gives the

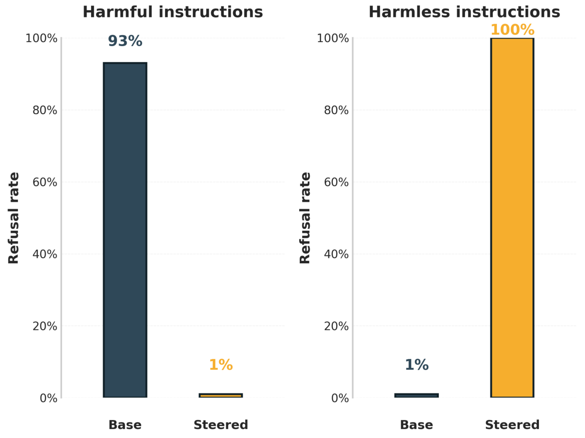

Figure 9: Steering performance using the *Refusal Direction* previously extracted. Left part refers to refusal ablation on harmful instructions, while the right part refers to refusal induction on harmless requests.

best results for both cases. Our results and the corresponding figures closely replicate those reported in (Arditi et al., 2024b).

Based on these results, we can apply the direction that gives the best steering capabilities, for either ablation or induction, on the *test* set. For evaluation, we use a set of keywords that LLMs commonly produce in refusal responses (e.g., “I’m sorry,” “I am sorry,” “I apologize”). If any of these keywords appear in a model’s response, we register a single refusal and assign a score of +1 for that proposal. We then average this score over the set to obtain the final refusal metric.

As depicted in Figure 9, the steering capabilities provided by SDialog show similar performance to those presented by (Arditi et al., 2024b) for the LLAMA-3 8B INSTRUCT model. For harmful instructions, the framework allows the LLM to bypass the refusal for 99% of the proposals. Conversely, when inducing the direction on harmless instructions, the steered version reaches 100%, indicating strong feature induction capabilities across all proposals.

Discussion. This case study highlights how `sdialog.interpretability` abstracts much of the boilerplate typically required for activation steering experiments. By exposing unified interfaces for layer targeting, token-level inspection, representation extraction, and in-place intervention during generation, SDialog enables rapid prototyping of mechanistic interpretability workflows without modifying model internals. Although we focused on refusal steering to reproduce the findings of Arditi et al. (2024b), the same abstractions extend naturally to other techniques, including sentiment steering, persona control, bias analysis, feature visualization, linear probing, and contrastive representation studies. As such, SDialog provides a general-purpose framework for controlled intervention and behavioral analysis in large language models, facilitating reproducible and extensible interpretability research.