Bel Esprit: Multi-Agent Framework for Building AI Model Pipelines

Yunsu Kim **AhmedElmogtaba Abdelaziz** **Thiago Castro Ferreira**
Mohamed Al-Badrashiny **Hassan Sawaf**
aiXplain, Inc.
Los Gatos, CA, USA
{firstname.lastname}@aixplain.com

Abstract

As the demand for artificial intelligence (AI) grows to address complex real-world tasks, single models are often insufficient, requiring the integration of multiple models into pipelines. This paper introduces *Bel Esprit*, a conversational agent designed to construct AI model pipelines based on user requirements. Bel Esprit uses a multi-agent framework where subagents collaborate to clarify requirements, build, validate, and populate pipelines with appropriate models. We demonstrate its effectiveness in generating pipelines from ambiguous user queries, using both human-curated and synthetic data. A detailed error analysis highlights ongoing challenges in pipeline building. Bel Esprit is available for a free trial at https://belesprit.aixplain.com[1].

1 Introduction

A single AI model is often insufficient for complex tasks, especially with multiple inputs or outputs, e.g., multimodal content moderation or multilingual video dubbing (Figure 1). Such tasks can be better addressed by integrating different models; by constructing a pipeline of interconnected models, we can automate intermediate steps and facilitate seamless task transitions. This approach, known as cascading models into a pipeline, has been widely used in applications like speech translation (Ney, 1999; Matusov, 2009) and voice conversion (Wu et al., 2018; Huang et al., 2020).

This paper presents *Bel Esprit*[2], a conversational assistant that implements sophisticated pipeline solutions composed of diverse AI models. Here are our main contributions:

- We formally define the task of model pipeline building as a graph generation problem involving scientific reasoning.

Query: I want to dub my video clip in French, German, and Spanish

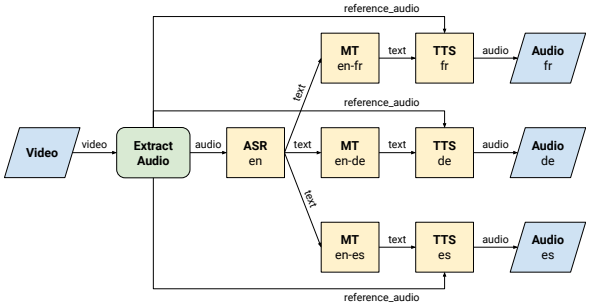


Figure 1: Query and model pipeline for multilingual video dubbing.

- We design a multi-agent framework that systematically enhances pipeline quality and alignment with user intent.
- We establish a rigorous evaluation scheme for pipeline building, including a data preparation protocol and automatic metrics.

2 Related Work

Automated Machine Learning Efforts to simplify machine learning for non-experts have focused on automating model selection (Kotthoff et al., 2017), neural architecture search (Jin et al., 2019; Zimmer et al., 2021), hyperparameter tuning (Bischl et al., 2023), and ensembling (Erickson et al., 2020; Shchur et al., 2023): mainly aiming to train a single model for atomic tasks. In contrast, Bel Esprit does not train models but assembles off-the-shelf models into pipelines, integrating various AI components for more complex tasks.

Agentic Workflow Generation Modern AI agents use multiple tools and subagents to break down complex tasks into subtasks and assign tools accordingly (Xi et al., 2023; Wang et al., 2024b). Existing workflow generation methods largely focus on writing LLM prompts for a few general agents or ordering simple utility functions, with

[1]Demo video: https://youtu.be/3KFSvrOPObY
[2]French for "beautiful mind"

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 329–339

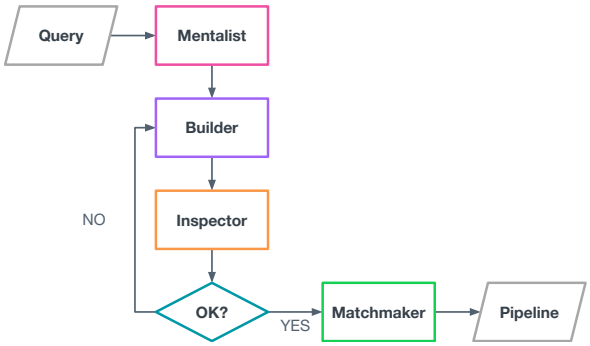


Figure 2: Agentic flow of Bel Esprit.

evaluations limited to classical reasoning tasks like math, coding, or QA (Chen et al., 2023; Zeng et al., 2023; Li et al., 2024; Zhuge et al., 2024; Zhang et al., 2024; Hu et al., 2024; Niu et al., 2025).

Bel Esprit expands this scope by integrating >70 AI functions across modalities (Appendix A) and devising tools for missing functionalities. It ensures pipeline reliability through conversational requirement clarification and formal graph-based verification. Also, the generated pipelines can serve as advanced tools within agents, reducing redundant planning and accelerating recurring tasks (Qian et al., 2023; Wang et al., 2024a; Cai et al., 2024).

3 Task Definition

Pipeline generation is a structured prediction task, where the input is a user query describing a computational task, and the output is a pipeline of AI functions to solve it. Each AI function may have parameters, e.g., language in speech recognition. The final output is basically a graph, with nodes representing inputs/outputs/functions, and edges denoting the data flow between them. To enhance the functionality of a pipeline, we introduce three special node types:

- **Router**: Directs the input data to different paths based on its modality.
- **Decision**: Sends data to different paths according to specific input values.
- **Script**: Executes an arbitrary task by running Python code.

Pipeline generation can be viewed as deductive reasoning where the AI functions exist as *premises* about data *entities* (Yu et al., 2023). Each premise conveys scientific knowledge from specific input to output. Given a user query as a new comprehensive *conclusion*, the objective is to find a reasoning path

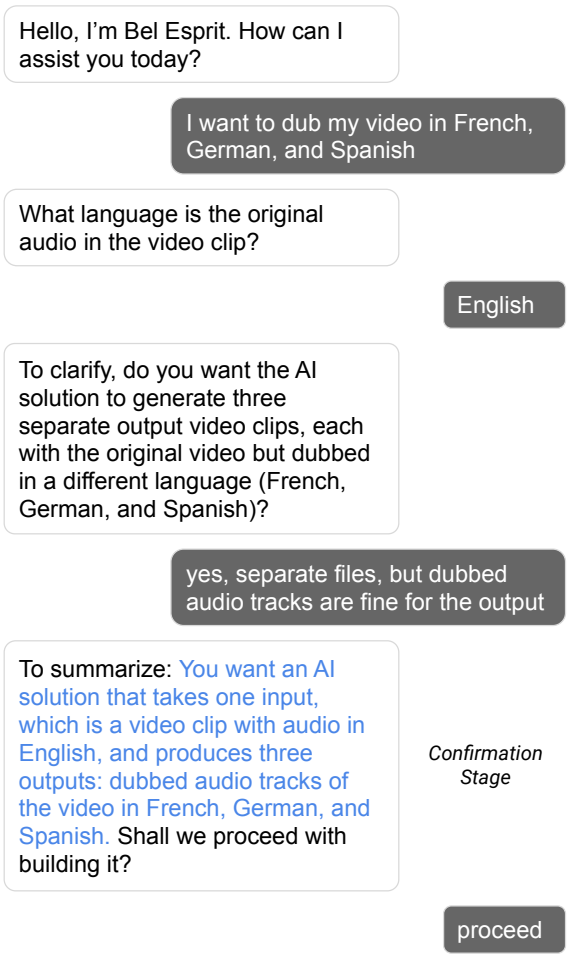


Figure 3: Example conversation between Mentalist and a user. The refined query is colored in blue.

comprising multiple premises (Saha et al., 2020; Creswell et al., 2022; Saparov and He, 2022).

4 Framework

In this work, we use an LLM to process user queries and generate pipeline structures through guided prompts. Instead of producing the pipeline in a single step, the framework follows a flow of multiple subagents (Figure 2). The process begins with *Mentalist*, followed by *Builder*, which creates an initial pipeline. This pipeline is then reviewed by *Inspector*. If the review fails, it loops back to Builder for revisions until an error-free pipeline is generated or the maximum iteration limit is reached. Once the pipeline passes inspection, it proceeds to *Matchmaker*, completing the final pipeline.

4.1 Mentalist

Mentalist is the agent responsible for interacting with the user and analyzing their requirements.

	Name	Modality	Language
Input	Video file	Video	English
Output	Audio track 1	Audio	French
	Audio track 2	Audio	German
	Audio track 3	Audio	Spanish

Table 1: Specification example.

4.1.1 Query Clarifier

User queries are often too ambiguous to build a correct solution. For example, they may lack detailed context, such as how "risk" is defined in a risk management system, or omit data properties, like the language of the input text. *Query Clarifier*, a chat interface, converts potentially ambiguous user queries into fully developed solution specifications. It identifies missing information and prompts the user to fill in the gaps. Once all necessary details are gathered, the system summarizes the conversation into a refined query that clearly outlines the solution's inputs and outputs, along with their modalities and relationships (Figure 3).

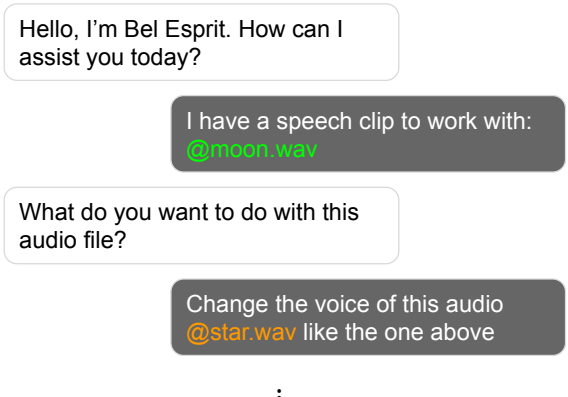
4.1.2 Specification Extractor

After the user confirms the clarified query, *Specification Extractor* extracts its technical details like name, modality, and required parameters for each input and output (Table 1). Such structured information offers clear guidance on which input and output nodes must be included, providing a strong foundation for constructing the intermediate flows; relying solely on long natural language queries often results in errors when building a solution.

4.1.3 Attachment Matcher

We found that many users begin by attaching a file, e.g., "I want to work with this text file to extract named entities and identify grammatical errors." Once a solution is generated, users need to know which input node in the pipeline graph corresponds to the attached file. While matching is straightforward for only a single input node, it becomes challenging when there are multiple input nodes, especially when some share the same modality.

In such cases, semantic analysis of the conversation is necessary to determine the specific characteristics of each input. Files may also be attached mid-conversation, with contextual clues before and after the attachment providing critical information for accurate matching. *Attachment Matcher* detects these associations and assigns each attached file to the appropriate input node. Note that file names themselves are not passed to the builder, as they may not be directly relevant to the solution.



	Name	Attachment
Input	Input speech	@star.wav
	Reference voice	@moon.wav
Output	Converted speech	N/A

Figure 4: Attachment matching example.

4.2 Builder

Builder constructs the pipeline graph based on the refined query (Section 4.1.1) and the extracted specification (Section 4.1.2). Builder is an LLM prompted with information on data types, function identifiers, node types, and graph constraints (Appendix B). Given the complexity of this task, a few example pipelines are included in the prompt to guide the generation process (Brown et al., 2020). Builder's output can be in any structured format, such as DOT or JSON.

4.2.1 Chain-of-Branches

Building a large graph in a single step is highly challenging. Generating token sequences in structured formats often leads to issues like hallucination and loss of consistency within the structure (Poesia et al., 2022; Beurer-Kellner et al., 2024; Tam et al., 2024). Inspired by the chain-of-thought (Wei et al., 2022b), we decompose the solution graph into distinct branches. Each branch represents a path from one or more input nodes to an output node; a pipeline with N output nodes will have N branches. These branches can be standalone solutions to subproblems derived from the user query. New branches can often reuse nodes from existing branches, reducing the number of totally new nodes to be generated for each branch.

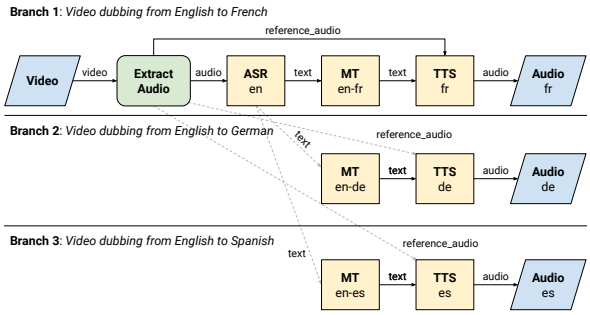
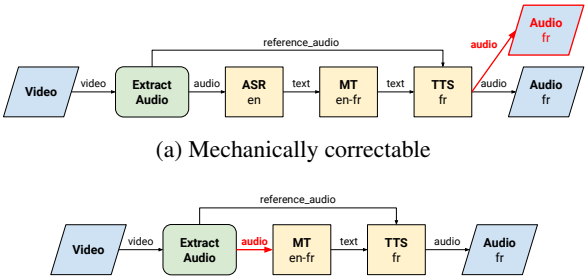


Figure 5: Example of generation using chain-of-branches. Gray dashed arrows indicate connections to previously generated nodes in existing branches. The final pipeline is in Figure 1.



(a) Mechanically correctable

(b) LLM-assisted correctable

Figure 6: Example of syntax errors (highlighted in red).

Figure 7: Example of semantic errors in a branch (highlighted in orange).

We prompt the LLM to generate one branch at a time, completing all nodes and edges for that branch before moving to the next (Figure 5). At each branch, we instruct the model to generate a brief comment to clarify the subproblem it addresses, ensuring the boundaries between branches.

4.3 Inspector

LLMs are particularly vulnerable to errors in scientific reasoning on lengthy contexts (Ahn et al., 2024; Ma et al., 2024). Even with a clarified query, errors may still occur due to the solution complexity. Similarly to critic models for LLM outputs (Ke et al., 2023; Xu et al., 2024; Gou et al., 2024), we developed *Inspector*, which analyzes the builder's output to identify errors in both the graph structure and semantic alignment with user requirements.

4.3.1 Syntax

First, we assess the structural integrity of the generated graph, independent of its intended function. We check violations of graph constraints (Appendix B), often due to improper node connections.

Some violations can be mechanically corrected immediately upon detection. Figure 6a illustrates such a case in generating Branch 1 of Figure 5. The output from a function node should connect to one output node, but multiple output nodes are linked to the same function output. This often arises when the user specifies multiple outputs in the solution. Such errors can be resolved by retaining only one output node and removing the duplicates.

Figure 6b illustrates an example where no simple correction is feasible. The machine translation (MT) node requires text input, yet audio extracted from a video input is routed directly to it. Resolving this modality mismatch involves either locating an existing node producing the necessary text output or creating a new node for the required transformation. Such complex corrections require an LLM to reconstruct the graph (Section 4.2).

4.3.2 Semantics

Next, we verify whether the graph semantically fulfills the user requirements. For each branch, we provide an LLM with a natural language summary that lists the nodes sequentially, outlining the path and its context within the pipeline. The LLM then identifies the corresponding requirements in the specification (Section 4.1.2) and flags any unmatched or missing steps in the branch path.

Figure 7 shows an example where the branch passes structural checks but fails in semantic alignment. In this case, the English transcription is routed directly to a French text-to-speech (TTS) node, assuming the same text modality suffices for synthesis; the builder overlooked the necessary translation step, resulting in a mismatch between the automatic speech recognition (ASR) output language and the intended TTS language.

4.4 Matchmaker

A pipeline from the Builder specifies only the data flow without assigning specific models to function nodes. *Matchmaker* gathers any additional information about the model selection in the query and finds the model that best align with the user's preferences, e.g., the latest MT model from Google or an ASR model specialized in medical domain. When no specific preference is provided, Matchmaker defaults to a predefined model choice.

Query: I want to understand English news clips more easily

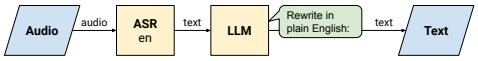


Figure 8: Example pipeline using a generic node.

Query: If I give you a summary, extend it to a long article; if it's an article, then summarize it.

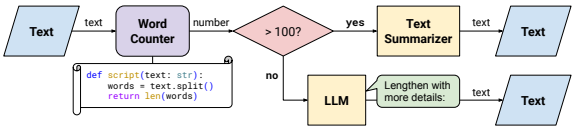


Figure 9: Example pipeline with a script node.

If a node requires a task for which no suitable model exists—often due to a complex user query or gaps in the platform's model library—Matchmaker employs the following fallback strategies.

4.4.1 Generic Nodes

Recent LLMs can perform generic tasks beyond their specific training when given a clear prompt (Brown et al., 2020; Wei et al., 2022a). For unavailable AI functions, we insert a custom LLM node with a prompt derived from the relevant part of the user query (Figure 8). This approach is useful for tasks like domain mixing or creative writing, where specialized models are scarce.

4.4.2 Script Generator

Some nodes are designated not for AI tasks but for simpler functions, such as counting words or extracting text from a PDF: a short script is sufficient (Figure 9). In such cases, we use an LLM to generate scripts; we begin by providing a script template that defines the input/output and their modalities, allowing the LLM to complete the method part based on the task description.

5 Experiments

To evaluate pipeline generation, we prepared query-pipeline pairs with evaluation metrics.

5.1 Data

Manual creation Given the high-level scientific nature of the task, we recruited five AI solution engineers at aiXplain, Inc. to create 82 realistic tasks and their corresponding pipelines. Each pipeline was then reviewed and, if necessary, revised by at least one other expert.

Structured synthesis with human correction To scale data collection, we automated the initial pipeline creation using rule-based expansion: nodes in a pipeline are expanded by adding others that match the input-output specifications. Starting with one or more input nodes, we constructed a tree-like structure that can branch into multiple output nodes. To manage complexity, we parameterized the number of AI function nodes and restrict each node to have a maximum of two children.

An LLM generates specifications and clear queries that enumerate the inputs and outputs. To simulate realistic user interactions, we then synthesized an initial user query by intentionally introducing ambiguity into the LLM prompt. In this way, we synthesized 500 data entries, retaining 359 after human review.

In total, we curated a dataset of 441 pipelines. For further details of the data, see Appendix C.

5.2 Metrics

Exact Match (EM) First, we count cases where the generated pipeline exactly matches the reference pipeline. Two nodes are considered a match if their types are identical and, if applicable, their functions and parameters are the same. For LLM nodes, we match prompts based on cosine similarity of their sentence embeddings, with a threshold of 0.5. For script nodes, we consider two code snippets a match if an LLM determines they perform the same task. Edges are matched if they connect the same source and target nodes with identical parameters. Determining such an exact match (EM) requires solving the graph isomorphism problem. To implement this, we adapted the VF2 algorithm (Cordella et al., 2004) to account for our problem.

Graph Edit Distance (GED) In our initial study, we found that many non-matching pipelines differ only slightly, typically by a single node or edge. Assigning a full penalty to such cases is too severe, as EM fails to capture incremental improvements. Therefore, we adopt graph edit distance (GED), which counts the number of edit operations—insertion, deletion, or substitution of nodes or edges—required to convert the generated graph to its reference. We apply the same matching conditions for nodes and edges as used in EM.

We used the depth-first GED algorithm (Abu-Aisheh et al., 2015) implemented in NetworkX (Hagberg et al., 2008). The edit operations have an equal weight of 1.0 for simplicity. We limited the

Framework setup	GPT-4o		Llama 3.1 405B		Llama3.1 70B	
	EM [%]	GED [%]	EM [%]	GED [%]	EM [%]	GED [%]
Builder	15.7	65.1	13.6	71.7	14.1	70.7
+ Query clarifier	25.1	44.4	21.5	52.8	19.0	54.4
+ Specification extractor	26.0	41.4	21.9	52.6	21.1	52.7
+ Chain-of-branches	25.2	40.3	21.9	52.6	19.0	53.9
+ Syntactic inspector	25.6	38.3	22.7	48.2	19.4	49.8
+ Semantic inspector	25.2	37.0	20.3	48.9	19.4	53.9

Table 2: Pipeline generation performance across framework configurations and Builder LLMs.

running time for each pipeline pair to 60 seconds on Macbook Pro 2023 (with M2 Pro).

5.3 Models

Mentalist's query clarifier (Section 4.1.1) and Builder (Section 4.2) utilize GPT-4o (OpenAI, 2024), while the rest of the framework, including data synthesis and evaluation, relies on the Llama 3.1 70B model (Dubey et al., 2024) when LLM assistance is required. Prompt similarity is computed using the all-MiniLM-L6-v2 model of Sentence Transformers (Reimers and Gurevych, 2019).

5.4 Results

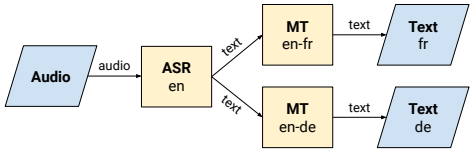
Table 2 shows the pipeline generation performance across various framework configurations. Starting with a baseline pipeline builder, we incrementally incorporate components from Mentalist, Builder, and Inspector, achieving +9.5% EM and -28.1% GED overall. For the Builder, GPT-4o outperforms open-source alternatives, with performance declining as model size decreases. Smaller models like Llama 3.1 8B yielded unacceptable performances, with EM rates below 3%.

Each component's contribution is evident in GED improvements for GPT-4o but less consistent for weaker models, while EM fails to capture the nuanced improvements. As a side note, semantic inspection occasionally confuses weaker Builders, leading to unnecessary graph repetitions and sporadic performance drops.

5.5 Qualitative Example

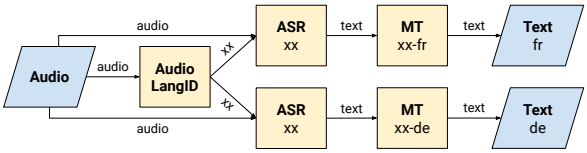
Figure 10 illustrates an example of incremental improvements in pipeline generation. The initial user query is ambiguous, as it does not specify the input language. The plain Builder assumes English as the input language and generates a pipeline accordingly. Mentalist refines the query to explicitly indicate that the input language is unknown, resulting in a pipeline that first performs language identification and passes the detected language to the ASR function.

Query: I want to translate my speech into French and German

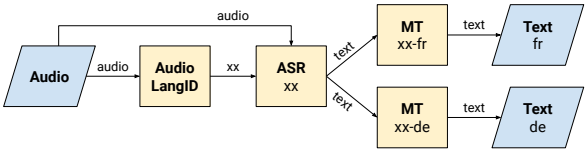


(a) Builder (plain)

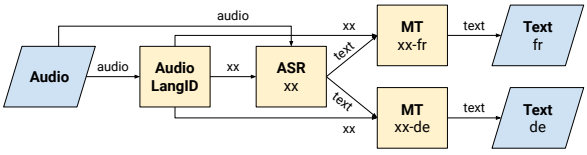
Refined Query: The requested solution takes speech in an unknown language as input and converts it to French text. The input language will be detected automatically.



(b) Mentalist + Builder (plain)



(c) Mentalist + Builder (chain-of-branches)



(d) Mentalist + Builder (chain-of-branches) + Inspector

Figure 10: Examples of generated pipelines across different framework configurations.

However, this version redundantly includes separate ASR nodes for French and German outputs. The chain-of-branches technique resolves this redundancy by generating one path at a time, enabling the reuse of the ASR node. Despite this improvement, the MT nodes lack source language parameters. The final configuration, which incorpo-

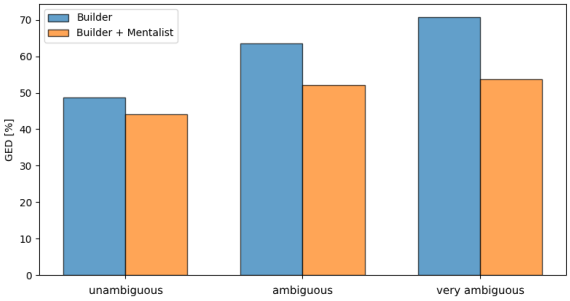


Figure 11: GED over increasing query ambiguity.

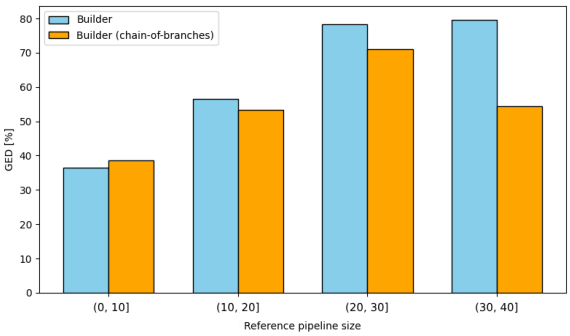


Figure 12: GED over increasing pipeline size.

rates Inspector, identifies this issue and adds edges from the language identifier to the MT nodes, producing a complete and correct pipeline.

6 Analysis

Ambiguity of query As shown above, ambiguity in user queries is a primary factor for poor pipeline generation performance. We used GPT-4o to rate the ambiguity of queries in three levels: unambiguous, ambiguous, and very ambiguous. Figure 11 shows performance computed for each level; pipeline generation becomes increasingly challenging with higher ambiguity. The Mentalist subagent significantly improves performance in such cases by clarifying missing information in queries and concretizing input and output requirements.

Pipeline size We also measured performance as a function of reference pipeline size, shown in Figure 12. As expected, larger pipelines——such as simultaneous processing of the same input across multiple paths—are more challenging to construct. However, the chain-of-branches technique proves to be effective in handling these cases by breaking the graph into manageable subgraphs.

Error Types We analyzed errors in generated pipelines using detailed logs of GED. Figure 13 shows that most errors stem from node substitutions, often due to parameter mismatches or incorrect node types (Figure 14).

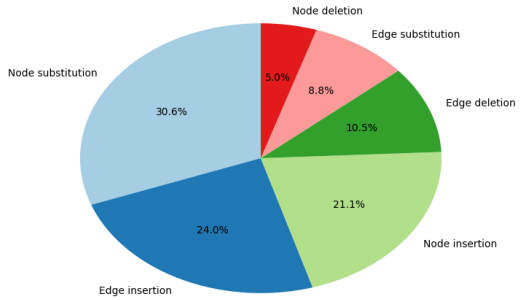


Figure 13: Distribution of edits required to align generated pipelines with reference pipelines.

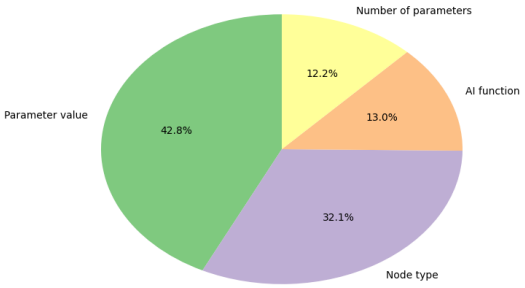


Figure 14: Causes for node substitution errors.

Node insertions occur when the builder fails to address all query requirements, often in large pipelines. Node deletions typically result from redundant function repetitions in separate paths. Both edits are also required when a misplaced node must be relocated to another path in the graph, which in turn needs corresponding edge insertions and deletions. These errors are generally less significant compared to node substitutions.

Edge errors often involve missing connections when a function require multiple inputs. While the Inspector can readily detect these, resolving them remains challenging as it requires comprehensive semantic understanding of the graph and query to locate the correct node supplying the missing data.

7 Conclusion

This paper introduces a novel task of generating AI solution pipelines from user queries and proposes Bel Esprit, a multi-agent framework consisting of Mentalist, Builder, and Inspector, which incrementally improve pipeline quality through query clarification, stepwise construction, and validation.

Future work includes employing retrieval-augmented generation (RAG) with a pool of valid pipelines and extending the framework to generate autonomous agents beyond static pipelines.

Limitations

Although the Mentalist (Section 4.1) enhances performance in ambiguous scenarios, the system still struggles with highly ambiguous queries, especially when critical input or output requirements are missing.

Pipeline building (Section 4.2) and matchmaking (Section 4.4) are restricted to a predefined pool of AI functions (Appendix A). Expanding this pool and incorporating their parameter details increases the prompt length, leading to higher computational costs. Generic nodes (Section 4.4.1) address this partially but are currently limited to text-to-text functions.

The Inspector (Section 4.3) does not verify the generated code for script nodes (Section 4.4.2), requiring custom test cases tailored to each script, which is not yet automated.

References

Zeina Abu-Aisheh, Romain Raveaux, Jean-Yves Ramel, and Patrick Martineau. 2015. An exact graph edit distance algorithm for solving pattern recognition problems. In *4th International Conference on Pattern Recognition Applications and Methods 2015*.

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. In *EACL Student Research Workshop*, pages 225–237.

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding llms the right way: Fast, non-invasive constrained generation. In *ICML*.

Bernd Bischl, Martin Binder, Michel Lang, Tobias Pielok, Jakob Richter, Stefan Coors, Janek Thomas, Theresa Ullmann, Marc Becker, Anne-Laure Boulesteix, et al. 2023. Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 13(2):e1484.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *NeurIPS*, volume 33, pages 1877–1901.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. Large language models as tool makers. In *ICLR*.

Guangyao Chen, Siwei Dong, Yu Shu, Ge Zhang, Jaward Sesay, Börje F Karlsson, Jie Fu, and Yemin Shi. 2023. Autoagents: A framework for automatic agent generation. *arXiv preprint arXiv:2309.17288*.

Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE TPAMI*, 26(10):1367–1372.

Antonia Creswell, Murray Shanahan, and Irina Higgins. 2022. Selection-inference: Exploiting large language models for interpretable logical reasoning. In *ICLR*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv:2407.21783*.

Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, et al. 2024. Critic: Large language models can self-correct with tool-interactive critiquing. In *ICLR*.

Aric A Hagberg, Daniel A Schult, and Pieter J Swart. 2008. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the Python in Science Conference*, pages 11–15. SciPy.

Shengran Hu, Cong Lu, and Jeff Clune. 2024. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*.

Wen-Chin Huang, Tomoki Hayashi, Shinji Watanabe, and Tomoki Toda. 2020. The sequence-to-sequence baseline for the voice conversion challenge 2020: Cascading asr and tts. In *Joint Workshop for the Blizzard Challenge and Voice Conversion Challenge 2020*, pages 160–164.

Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-keras: An efficient neural architecture search system. In *KDD*, pages 1946–1956.

Pei Ke, Bosi Wen, Zhuoer Feng, Xiao Liu, Xuanyu Lei, Jiale Cheng, Shengyuan Wang, Aohan Zeng, Yuxiao Dong, Hongning Wang, et al. 2023. Critiquellm: Scaling llm-as-critic for effective and explainable evaluation of large language model generation. *arXiv:2311.18702*.

Lars Kotthoff, Chris Thornton, Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. 2017. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *JMLR*, 18(25):1–5.

Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. 2024. Autoflow: Automated workflow generation for large language model agents. *arXiv preprint arXiv:2407.12821*.

Yubo Ma, Zhibin Gou, Junheng Hao, Ruochen Xu, Shuohang Wang, Liangming Pan, Yujiu Yang, Yixin Cao, Aixin Sun, Hany Awadalla, et al. 2024. Sciagent: Tool-augmented language models for scientific reasoning. *arXiv:2402.11451*.

Evgeny Matusov. 2009. *Combining Natural Language Processing Systems to Improve Machine Translation of Speech*. Ph.D. thesis, RWTH Aachen University.

Hermann Ney. 1999. Speech translation: Coupling of recognition and translation. In *ICASSP*, volume 1, pages 517–520.

Boye Niu, Yiliao Song, Kai Lian, Yifan Shen, Yu Yao, Kun Zhang, and Tongliang Liu. 2025. Flow: Modularized agentic workflow automation. In *ICLR*.

OpenAI. 2024. Gpt-4o system card. *arXiv:2410.21276*.

Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. In *ICLR*.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *EMNLP Findings*, pages 6922–6939.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *EMNLP*. Association for Computational Linguistics.

Swarnadeep Saha, Sayan Ghosh, Shashank Srivastava, and Mohit Bansal. 2020. Prover: Proof generation for interpretable reasoning over rules. In *EMNLP*, pages 122–136.

Abulhair Saparov and He He. 2022. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. In *ICLR*.

Oleksandr Shchur, Ali Caner Turkmen, Nick Erickson, Huibin Shen, Alexander Shirkov, Tony Hu, and Bernie Wang. 2023. Autogluon–timeseries: Automl for probabilistic time series forecasting. In *International Conference on Automated Machine Learning*, pages 9–1. PMLR.

Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung-yi Lee, and Yun-Nung Chen. 2024. Let me speak freely? a study on the impact of format restrictions on performance of large language models. *arXiv:2408.02442*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024a. Voyager: An open-ended embodied agent with large language models. *TMLR*.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024b. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186–345.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022a. Emergent abilities of large language models. *TMLR*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022b. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, volume 35.

Yichiao Wu, Patrick Lumban Tobing, Tomoki Hayashi, Kazuhiro Kobayashi, and Tomoki Toda. 2018. The nu non-parallel voice conversion system for the voice conversion challenge 2018. In *The Speaker and Language Recognition Workshop (Odyssey 2018)*, pages 211–218.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv:2309.07864*.

Wenda Xu, Daniel Deutsch, Mara Finkelstein, Juraj Juraska, Biao Zhang, Zhongtao Liu, William Yang Wang, Lei Li, and Markus Freitag. 2024. Llmrefine: Pinpointing and refining large language models via fine-grained actionable feedback. In *NAACL Findings*, pages 1429–1445.

Fei Yu, Hongbo Zhang, and Benyou Wang. 2023. Natural language reasoning, a survey. *arXiv preprint arXiv:2303.14725*.

Zhen Zeng, William Watson, Nicole Cho, Saba Rahimi, Shayleen Reynolds, Tucker Balch, and Manuela Veloso. 2023. Flowmind: automatic workflow generation with llms. In *Proceedings of the Fourth ACM International Conference on AI in Finance*, pages 73–81.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. 2024. Aflow: Automating agentic workflow generation. *arXiv preprint arXiv:2410.10762*.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. Gptswarm: Language agents as optimizable graphs. In *ICML*.

Lucas Zimmer, Marius Lindauer, and Frank Hutter. 2021. Auto-pytorch: Multi-fidelity metalearning for efficient and robust autodl. *IEEE transactions on pattern analysis and machine intelligence*, 43(9):3079–3090.

Text	Image	Audio
Translation	Image Captioning	Speech Recognition
Summarization	Optical Character Recognition	Speech Synthesis
Text Generation	Document Extraction	Voice Cloning
Text Transformation	Image Generation from Text	Audio Forced Alignment
Question Answering	Image-to-Image Translation	Audio Generation
Text Classification	Image Manipulation	Audio-to-Audio Translation
Topic Classification	Image Classification	Subtitling
Sentiment Analysis	Image Expression Detection	Multilingual Subtitling
Emotion Detection	Object Detection	ASR Quality Estimation
Language Identification	Image Content Moderation	Audio Transcript Analysis
Text Spam Detection	Visual Question Answering	Audio Transcript Improvement
Offensive Language Identification	Depth Estimation	Audio Classification
Text Content Moderation	Image Segmentation	Audio Language Identification
Token Classification	Mask Generation	Audio Speaker Diarization
Named Entity Recognition	Image Compression	Voice Activity Detection
Entity Linking	Image Embedding	Speech Classification
Entity Sentiment Analysis	**Video**	Speech Embedding
Coreference Resolution	Video Generation from Text	**Tabular**
Syntactic Parsing	Video Generation from Image	Tabular Classification
Semantic Parsing	Viseme Generation	Tabular Captioning
Slot Filling	Extract Audio From Video	Tabular Regression
Text Normalization	Video Speaker Diarization	Table Question Answering
Text Denormalization	Video Classification	Time Series Forecasting
Diacritization	Video Label Detection	**Others**
Text Embedding	Video Content Moderation	Similarity Search
	Video Expression Detection	Model Likelihood

Table 3: AI functions used in Bel Esprit, categorized by their primary modality.

A List of AI Functions

AI functions in Table 3 are considered as possible nodes of a pipeline in this work.

B Graph Constraints

Nodes

- An input node should have no previous nodes
- An input node should have only one output parameter
- An output node should have no next nodes
- There should be no multiple output nodes with the same incoming link
- A router node should have a single input node as its predecessor
- A router node should have two or more output parameters, each of which has a different modality
- A router node should not be connected with another router node
- A function name should exist in the predefined list of functions
- Parameters of a function node should exist in the predefined list of parameters
- A function node should have all its required input parameters

Edges

- An input parameter should have only one incoming edge
- An output parameter should have at least one outgoing edge if it is not an output node
- Every node should be reachable from an input node
- An edge should connect existing parameters
- The connected parameters should have the same modality

C Query-Pipeline Dataset

Domain Coverage The dataset demonstrates strong coverage of practical applications across various domains (Figure 15):

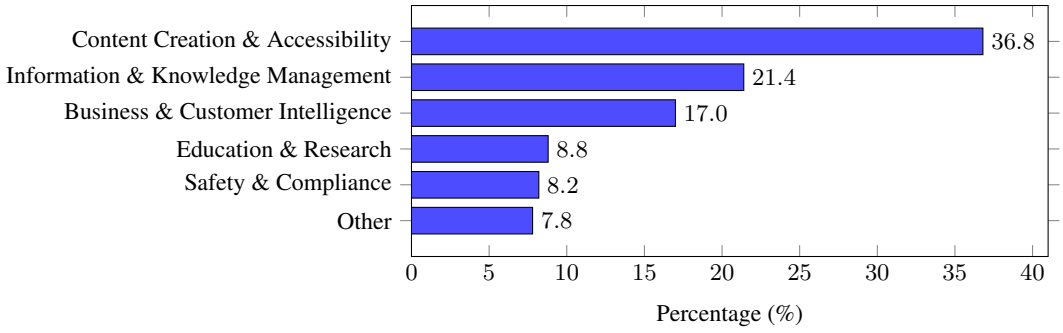


Figure 15: Distribution of applications domains of the data entries.

- **Business & Customer Intelligence**: Analyze company documents or customer feedbacks to gain business insights.
    - *I'm looking for a solution that can identify and categorize customer feedback into different themes, such as product quality, customer service, and delivery experience.*
- **Content Creation & Accessibility**: Enhance content accessibility across languages and modalities.
    - *I am looking for a solution to convert my French book into an audiobook in the original language as well as in English, Spanish, and Portuguese.*
- **Information & Knowledge Management**: Extract structured information from unstructured data.
    - *How to generate a 10K rows high-quality Modern Standard Arabic (MSA) corpus for sentiment analysis from an unlabelled text format English dataset?*
- **Safety & Compliance**: Conduct content moderation and safety applications.
    - *I need a pipeline that can detect and redact sensitive information like personal identifiers from texts, audios, and videos.*
- **Educational & Research**: Assist students or generate educational materials.
    - *I need a pipeline to assess the readability of documents. The documents are in various languages. Please also provide suggestions for simplification.*

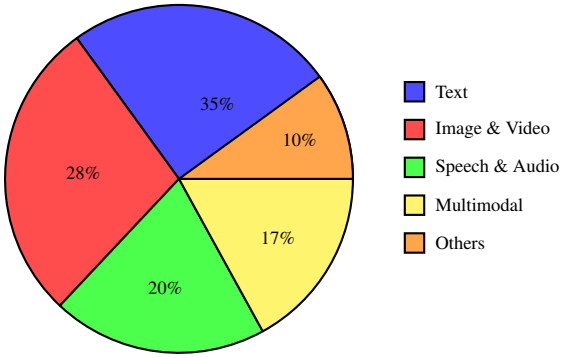


Figure 16: Distribution of modalities involved across the data entries.

Input \ Output	Text	Audio	Image	Video
Text	25%	18%	12%	8%
Audio	15%	10%	5%	3%
Image	14%	7%	12%	4%
Video	10%	6%	5%	7%

Figure 17: Heatmap of modality conversions in the dataset's reference pipelines.

Modality Coverage We categorize the task modality of each dataset entry in Figure 16. The "Image & Video" and "Speech & Audio" categories include basic transformations to and from text, such as speech recognition. The "Multimodal" category represents more advanced integrations involving multiple modalities, such as functions that process both image and text inputs.

Figure 17 illustrates the frequency of modality conversions required to solve the queries in the dataset, showing that all types of transformations between the four modalities are well covered.