# Inter-IIT Tech Meet 12.0: Final Report
# DevRev Team - 69


## Abstract

This paper tackles the challenge of aligning a query with a set of tools. The problem at hand can be further divided into two sub-problems. Given a user query, which is a natural language text, the first task is to extract all the possible tools that will be required to answer that query, and the second one is that for each tool, its arguments need to be populated, i.e., argument's value extraction. Both these tasks have their own challenges. For instance, with the second task, given an argument for a tool, its value may not be present in the current query, i.e., it may come from the execution of a previous tool. Another problem can occur in the tool extraction step itself, like how to associate one query with multiple tools that may be required to resolve it. All these problems and more are explored and resolved in our proposed solution strategy. We have created a robust query-tool matching system.


## 1 Background Research

Firstly, we took a high-level view of the problem statement and brainstormed about potential methods that could be used to tackle it. We first discussed some learning approaches, which would require training of deep learning model, but this presented itself with a challenge, which is to train any deep neural network, we would require data. The data provided to us was not sufficient to train a model. Another potential problem was the dynamic nature of the data on tools. This could potentially change, and thus, a learning approach was not suitable for such a dynamic, data-scarce problem statement. Hence, we discarded the learning approach and delved into some non-learning approaches.

We came up with an embedding-matching mechanism that could potentially provide us with a ground to work on. After performing a bunch of preliminary experiments, like matching embeddings of queries to that of tool descriptions, this did not yield extraordinary results, but it was enough to get us working in this direction. As we were dissecting the problem further and further, more nuances emerged that needed to be addressed. Some major ones that affected the wider direction of our final solution strategy are discussed below.

A. How to handle one query corresponding to multiple tools?

   Initially, we began by matching the embeddings of our query to all the tools' descriptions, which led to another problem: how do we select which tools actually go into the solution? Do we set a threshold and pick the top $k$ tools, but then how do we decide this $k$ for each query? A better solution we thought would work is splitting the query into multiple sub-queries and then resolving each sub-query, with the assumption that each sub-query is distilled enough to match with just one tool. This strategy worked with a lot of our test cases, and it also gave more pronounced similarity matches because the density of information required to match to a tool per sub-query increased. Now, the information is more concentrated within one sub-query, which earlier was diluted. But it came with another problem. If the top $k$ tools (ordered by the cosine similarity scores) have very close similarity scores, for instance, 0.501, 0.499, 0.490, then it isn't necessarily wise to pick the first because the difference is not very apparent and this could lead to a lot of near misses.

B. How to be more robust in tool selection, resolve conflicts, and avoid near misses?

   We propose an ensemble learning approach, which is applied in a non-learning setting. This is one of the novelties in our approach. We decided to use multiple models for generating embeddings

and capturing the information per sub-query in different ways. For instance, we experimented with HuggingFace (a)'s BERT and OpenAI's embedding models and checked the similarity score between the embedding of the same query passed through these two models. The similarity score we observed was much less than one, which indicates that the way these two models capture information from a sentence is quite different, and it may reveal subtleties in the query itself. After passing the query through multiple embedding models instead of one and performing cosine-similarity matching with the toolset, we performed a simple aggregation, taking a simple majority vote and deciding which tool should be associated with each sub-query. This resolves the problem of near-misses and conflicts because if a majority of the models are able to capture the subtle 0.01 magnitude of difference, then it is highly likely that this difference is considerable and not just a bias in one particular model.

C. How did we go about argument extraction?

   After successfully extracting the correct tools per sub-query, we decided to use another embedding matching approach. We tried matching embeddings of argument descriptions with sub-queries (here, we used word embeddings and matched every word of the sub-query with the arguments' descriptions embedding). This did not yield the results we hoped for, so we came up with another approach. This is perhaps one of the most important novelties in our solution. We decided to use a question-answering model to extract argument values. We worked on designing a prompt for the Q&A Model, and this approach yielded much better results. But what If the argument value does not exist in the current local context of the sub-query but is actually a result of a previous tool execution of another sub-query within the same query?

D. How did we go about handling dependency in argument extraction from previous tool executions?

   The approach was to combine previous sub-queries with the local context of the current sub-query and then feed this to the Q&A Model to extract arguments. Our approach also involves carefully tuning multiple thresholding parameters after meticulously performed experiments. This resolved the issue.

E. After we have the tools and queries, how did we compose tools?

   We spent a significant time researching ways to tackle this problem. The solution needs to take into account the dependency between the tools and use that to construct the final composition of tools and arguments. We experimented with maintaining a knowledge graph and using it to order the tools. (Almost like topological sorting). However, the very construction of such a knowledge graph would require prior dependency between tools. This was not given to us, but we did try to approach it, and we were able to come up with an approach to generate the graph, but it did not yield any information that we could use concretely for the composition of tools. Instead, we modified our argument extraction step to get the right composition of tools with as much accuracy as we could.

Throughout building the solution, we did extensive research into strategies to handle our problems at hand in the best way possible. We did not get concrete solutions in the literature or our research online, but it did give us many hints, and we proceeded to brainstorm extensively to come up with novelties that could tackle our problems at hand. We proposed the final solution with all the discussions above in mind, the details of which will be discussed in later sections.

## 2 Methodology

We segmented the task of identifying the tools necessary to address user queries into four distinct sub-parts.

### 2.1 Tool Set Processing

To initiate the toolset processing, we began by transforming the provided tool descriptions table into a JSON file. This file served as the basis for extracting the description of each tool, which was then subjected to

various sentence embedding models. The resulting embeddings, along with those obtained from processing the arguments, were systematically stored in a '.npy' file. The embedding process can be expressed mathematically as follows:

$$T_{ij}^{e} = M_j(T_i)$$
$$A_{ij}^{e} = M_j(A_i)$$

Here, $T_i$ represents the description of the $i^{th}$ tool, $M_j$ denotes the $j^{th}$ sentence embedding model, and $T_{ij}^{e}$ signifies the sentence embedding of $T_i$ calculated using the model $M_j$.

### 2.2 Query Processing

For the query processing stage, we created the split function $S$, which employs the spaCy module to partition the query into individual sub-queries. Subsequently, we compute the sentence embedding for each sub-query $q_i$ using the same models employed in the toolset, resulting in $q_{ij}^{e}$.

$$[q_1, q_2, ..., q_k] = S(Q)$$
$$q_{ij}^{e} = M_j(q_i)$$

Here, $q_1, q_2, ..., q_k$ are sub-queries of query $Q$, $q_i$ denotes the $i^{th}$ sub-query and $q_{ij}^{e}$ signifies the sentence embedding of $q_i$ calculated using $M_j$

### 2.3 Tool Extraction

In the tool extraction phase, we computed the cosine similarity of each sub-query embedding with every tool description embedding, recording the results in a table, as illustrated in Figure 1. Similar tables were generated using multiple sentence-transformer embedding models (can be found at HuggingFace (b)) based on the work of Reimers & Gurevych (2020). Subsequently, we identified the tool associated with the maximum cosine similarity for each sub-query in each table, leading to the formation of the table depicted in Figure 8. To determine the final tool for each sub-query, a voting mechanism was applied across models (names of models used can be found in Appendix A):

$$a_{ij} = CosSim(q_{jn}^{e}, T_{in}^{e})$$
$$A_{nk} = max_k(a_{ik})$$
$$T_f = votek(Ank)$$

In this context, $CosSim$ represents the cosine similarity function, calculating the cosine value between two embeddings. $max_k$ is a function that identifies the maximum values over the $k^{th}$ column, and $vote_k$ involves a voting process to determine the tool with the highest occurrence over the $k^{th}$ column.

### 2.4 Getting Argument Values

For obtaining the argument values, we employ a BERT-based Question-Answering (Q&A) model . As depicted in 9, we input the augmented sub-query as context, wherein $\varepsilon$ represents a custom-designed string crafted through prompt engineering, along with a question $\mu$ formulated to extract argument values from the customized context.

$$\text{ArgVal}_i = \text{Answer}(q_i + \varepsilon, \mu)$$

### 2.5 Generating Response

By utilizing the tools, arguments, and values obtained through the aforementioned steps, we construct a structured JSON response that represents the anticipated answer.

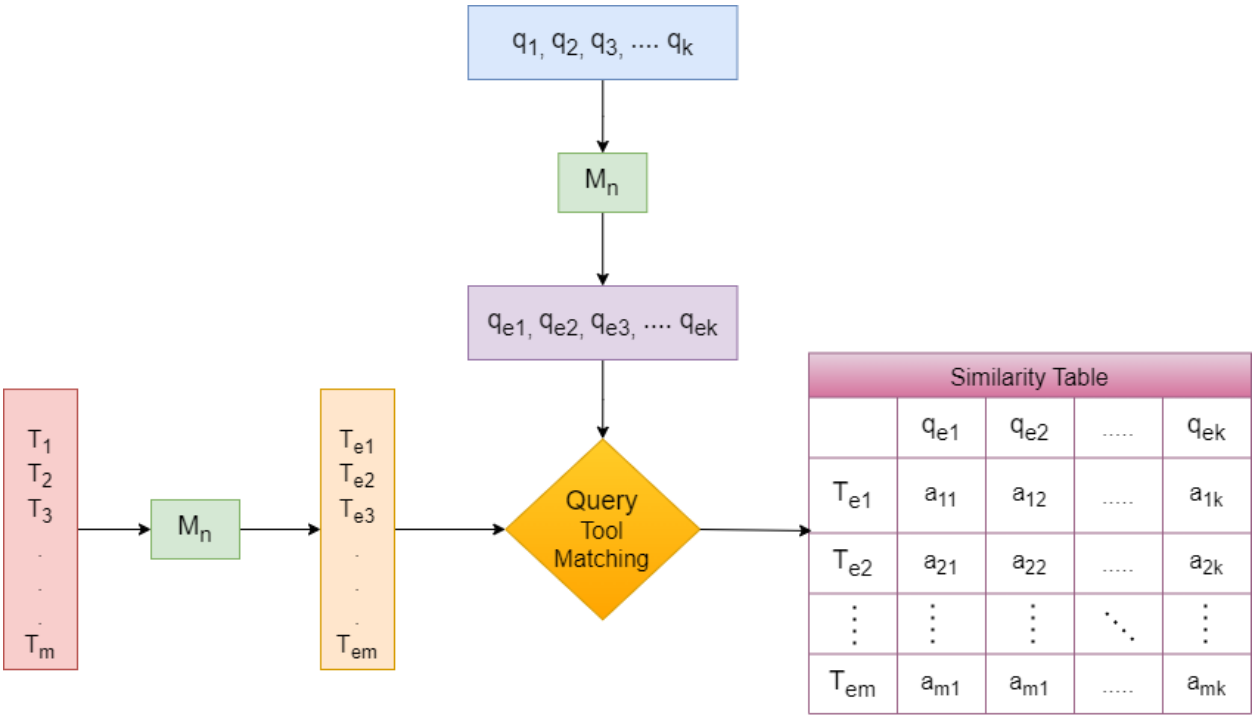


Figure 1: First, we calculate the embeddings of tool description and sub-queries. These embeddings are passed through a Query Tool Matching (cosine similarity) function to calculate the similarity between tools and sub-queries.

| Aggregated Similarity Table | | | | |
|---|---|---|---|---|
| | $q_{e1}$ | $q_{e2}$ | ..... | $q_{ek}$ |
| $M_1$ | $A_{11}$ | $A_{12}$ | ..... | $A_{1k}$ |
| $M_2$ | $A_{21}$ | $A_{22}$ | ..... | $A_{2k}$ |
| ⋮ | ⋮ | ⋮ | ⋱ | ⋮ |
| $M_m$ | $A_{m1}$ | $A_{m2}$ | ..... | $A_{mk}$ |

Figure 2: We will get an aggregated similarity table after taking the maximum tool corresponding to similarity values for each sub-query for each table.

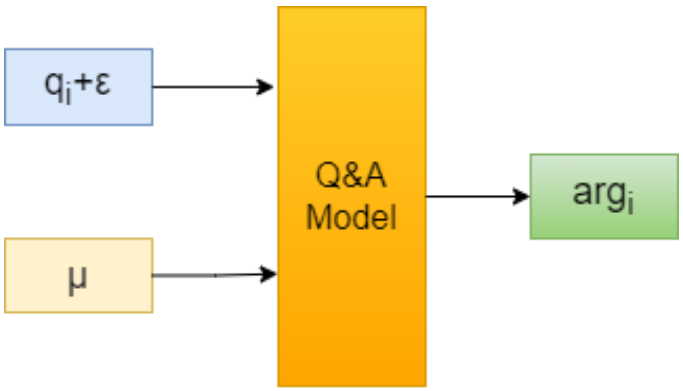


Figure 3: Passing the context($q_i + \varepsilon$) and the question($\mu$) to the BERT-based Question-Answering model to get the argument value.

## 3 Experimentation

### 3.1 Sub-query Generation

We used the SpaCy module to process the user's query. The query $Q$ serves as the input and is decomposed into smaller, meaningful sub-queries by identifying natural delimiters, such as punctuation and conjunctions.

The output is a list of these sub-queries $[q_1, q_2, ..., q_k]$. For instance, consider as following:

Input : 'Get all work items similar to TKT-123, summarize them, create issues from that summary, and prioritize them'

Output: ['Get all work items similar to TKT-123', 'summarize them', 'create issues', 'that summary', 'prioritize them']

### 3.2 Semantic Similarity Calculation

Using the `SentenceTransformer` module, we transform a tool description, $T$, and a sub-query, $q$, into their respective embeddings, $T_e$ and $q_e$. The output is the cosine similarity, $a$, between these two embeddings. As an illustration:

Inputs : $q$ : 'Get all work items similar to TKT-123'
$T$ : 'Returns a list of work items that are similar to the given work item'

Output : $a$ : 0.6025

### 3.3 Question-Answering Model

Once we have identified the final tool required for each sub-query, we use a BERT-based Question-Answering (Q&A) model to extract the argument values.

Consider the sub-query, $q$ : 'Get all work items similar to TKT-123'

The tool identified for this task is `get_similar_work_items` with an argument `work_id`. We construct a context and a question as:

$q + \varepsilon$ : 'Get all work items similar to TKT-123 given tool used get_similar_work_items'

$\mu$ : 'What is the argument value for the work_id parameter in the get_similar_work_items tool?'

This context and question are input into our Q&A model. The model's output will be the argument value.

Output : TKT-123

## 4 Results and Discussion

The performance of the system is evaluated using the BLEU (Bilingual Evaluation Understudy) metric. Our system achieved an accuracy of 95%. Results can be found in Appendix B. However, there are some instances where the performance is less than optimal. This might be attributed to the fact that the system currently lacks knowledge of inter-tool dependencies, leading to inaccuracy in tool extraction. Besides, some tools are associated with only single words, which presents a challenge as our system primarily works on sentence-level sub-queries. These challenges highlight the areas of future improvement in our solution. The results on the sample test cases can be found in Appendix B.

## 5 Conclusion

We have successfully developed a robust query-tool matching system. The tool extraction module and the argument extraction module have been effectively implemented. We have also outlined robust aggregation strategies for tool selection. Despite the challenges in extracting values of complex arguments, our use of a BERT-based Question-Answer model has shown accuracy close to 95%. In conclusion, we have devised a resilient strategy and introduced numerous novel approaches to address various challenges encountered in developing our innovative and efficient solution.

## 6 Future Work

In future work, two key strategies could be explored to enhance our solution. First, a knowledge graph to keep a structured representation of tool dependencies and relationships could be maintained, thereby enhancing the precision of tool selection. Second, while our current methodology uses sentence-level sub-queries, there is a scope to explore word-level sub-queries. A sliding window approach, taking windows of varying sizes, could prove particularly beneficial for the extraction of tools that are associated with just one word. The implementation of these strategies could lead to improvements in the performance of our system.

## References


HuggingFace. Bert. `https://huggingface.co/docs/transformers/main/model_doc/bert`, a. Date Accessed: 12 Dec 2023.

HuggingFace. Sentence transformers. `https://huggingface.co/sentence-transformers`, b. Date Accessed: 12 Dec 2023.

OpenAI. Embeddings. `https://platform.openai.com/docs/guides/embeddings`. Date Accessed: 12 Dec 2023.

Nils Reimers and Iryna Gurevych. Making monolingual sentence embeddings multilingual using knowledge distillation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2020. URL `https://arxiv.org/abs/2004.09813`.

SpaCy. Sentence splitting. `https://spacy.io/usage`. Date Accessed: 12 Dec 2023.


## A Appendix

List of Sentence Transformer Models used:

1. all-MiniLM-L6-v2
2. sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2
3. sentence-transformers/msmarco-MiniLM-L6-cos-v5
4. sentence-transformers/paraphrase-MiniLM-L3-v2
5. sentence-transformers/all-mpnet-base-v2
6. sentence-transformers/all-MiniLM-L6-v2
7. sentence-transformers/multi-qa-MiniLM-L6-cos-v1
8. sentence-transformers/paraphrase-albert-small-v2
9. sentence-transformers/all-distilroberta-v1
10. sentence-transformers/multi-qa-mpnet-base-cos-v1
11. sentence-transformers/msmarco-distilbert-dot-v5
12. sentence-transformers/paraphrase-TinyBERT-L6-v2
13. sentence-transformers/nli-bert-large
14. sentence-transformers/LaBSE
15. sentence-transformers/clip-ViT-L-14

## B Appendix

```
[
    {
      "tool_name": "get_similar_work_items",
      "arguments": [
        {
          "argument_name": "work_id",
          "argument_value": "TKT-123"
        }
      ]
    },
    {
      "tool_name": "summarize_objects",
      "arguments": [
        {
          "argument_name": "objects",
          "argument_value": "$$PREV[0]"
        }
      ]
    },
    {
      "tool_name": "summarize_objects",
      "arguments": [
        {
          "argument_name": "objects",
          "argument_value": "$$PREV[1]"
        }
      ]
    },
    {
      "tool_name": "prioritize_objects",
      "arguments": [
        {
          "argument_name": "objects",
          "argument_value": "$$PREV[2]"
        }
      ]
    }
]
```


Figure 4: Our output for query "Get all work items similar to TKT-123, summarize them, create issues from that summary, and prioritize them"


```
[
    {
    "tool_name": "create_actionable_tasks_from_text",
    "arguments": [
            {
    "argument_name": "text",
    "argument_value": "T"
            }
        ]
    },
    {
    "tool_name": "get_sprint_id",
    "arguments": []
    },
    {
    "tool_name": "add_work_items_to_sprint",
    "arguments": [
            {
    "argument_name": "work_ids",
    "argument_value": "$$PREV[0]"
            },
            {
    "argument_name": "sprint_id",
    "argument_value": "$$PREV[1]"
            }
        ]
    }
]
```


Figure 5: Reference for query "Get all work items similar to TKT-123, summarize them, create issues from that summary, and prioritize them"

```
[
  {
    "tool_name": "create_actionable_tasks_from_text"
    "arguments": [
      {
        "argument_name": "text",
        "argument_value": "$$PREV[0]"
      }
    ]
  },
  {
    "tool_name": "add_work_items_to_sprint",
    "arguments": [
      {
        "argument_name": "work_ids",
        "argument_value": "$$PREV[1]"
      },
      {
        "argument_name": "sprint_id",
        "argument_value": "$$PREV[2]"
      }
    ]
  }
]
```


Figure 6: Our output for query "Given a customer meeting transcript T, create action items and add them to my current sprint"


```
[
  {
    "tool_name": "create_actionable_tasks_from_text"
    "arguments": [
      {
        "argument_name": "text",
        "argument_value": "$$PREV[0]"
      }
    ]
  },
  {
    "tool_name": "add_work_items_to_sprint",
    "arguments": [
      {
        "argument_name": "work_ids",
        "argument_value": "$$PREV[1]"
      },
      {
        "argument_name": "sprint_id",
        "argument_value": "$$PREV[2]"
      }
    ]
  }
]
```


Figure 7: Reference for query "Given a customer meeting transcript T, create action items and add them to my current sprint"

```
[
  {
    "tool_name": "summarize_objects",
    "arguments": [
      {
        "argument_name": "objects",
        "argument_value": "$$PREV[0]"
      }
    ]
  }
]
```


Figure 8: Our output for query “What are my all issues in the triage stage under part FEAT-123? Summarize them.”

```
[
  {
    "tool_name": "whoami",
    "arguments": []
  },
  {
    "tool_name": "works_list",
    "arguments": [
      {
        "argument_name": "stage.name",
        "argument_value": [
          "triage"
        ]
      },
      {
        "argument_name": "applies_to_part",
        "argument_value": [
          "FEAT-123"
        ]
      },
      {
        "argument_name": "owned_by",
        "argument_value": [
          "$$PREV[0]"
        ]
      },
      {
        "argument_name": "type",
        "argument_value": [
          "issue"
        ]
      }
    ]
  },
  {
    "tool_name": "summarize_objects",
    "arguments": [
      {
        "argument_name": "objects",
        "argument_value": "$$PREV[1]"
      }
    ]
  }
]
```


Figure 9: Reference for query “What are my all issues in the triage stage under part FEAT-123? Summarize them.”