CLaRA: Cost-effective LLMs Function Calling Based on Vector Database

Miloslav Szczypka[1,✉], **Lukáš Jochymek**[1,✉], **Alena Martinková**[1,✉]
[1]Rankacy, Tovární 589, Ostrava, Czech Republic
info@rankacy.com✉

Abstract
Since their introduction, function calls have become a widely used feature within the OpenAI API ecosystem. They reliably connect GPT's capabilities with external tools and APIs, and they quickly found their way into the other LLMs. The challenge one can encounter is the number of tokens consumed per request since each definition of each function is always sent as an input. We propose a simple solution to effectively decrease the number of tokens by sending only the function corresponding to the user question. The solution is based on saving the functions to the vector database, where we use the similarity score to pick only the functions that need to be sent. We have benchmarked that our solution can decrease the average prompt token consumption by 210% and the average prompt (input) price by 244% vs the default function call. Our solution is not limited to specific LLMs. It can be integrated with any LLM that supports function calls, making it a versatile tool for reducing token consumption. This means that even cheaper models with a high volume of functions can benefit from our solution.

Keywords: *Function Calls, Large Language Models, OpenAI, GPT, Vector Database.*

Received: 13 January 2024
Accepted: 10 June 2024
Online: 30 June 2024
Published: 15 July 2024

1 Introduction

Function calls, introduced with the OpenAI API update on June 13, 2023, have revolutionized the integration of GPT's capabilities with external tools and APIs [1]. These function calls enable the model to use external, user-defined functions. However, this powerful functionality has its challenges. The current system, where each function definition in the API call contributes to the overall token count, can be inefficient and costly due to the high number of tokens consumed per request, the impact of this can be seen in Table 3.

To address this challenge, we propose a straightforward yet effective solution that reduces the number of tokens consumed by sending only the relevant function definitions based on the user's query. Our approach involves storing functions in a vector database and utilizing similarity scores to select only the necessary functions for each request. Benchmarking this solution has shown a significant decrease in token consumption and input costs.

2 Problem statement

In implementing function calls within transformer-based models such as GPT-3 [3] or GPT-4 [10], a significant operational challenge arises from the high consumption of input tokens. Each function call contributes to the token count, including its name, parameters, and the required syntactical structure (formatted as JSON). This can scale to the point where the whole context window comprises the functions. This becomes particularly problematic as OpenAI's API imposes a strict token limit per request, which includes both the user's input and the model's output. When multiple function calls or complex parameters are necessary for a single conversation turn, a substantial portion of the available tokens is consumed, thereby reducing the tokens available for maintaining conversational context and handling extensive user inputs. This limitation constrains the depth and fluidity of dialogues and escalates operational costs, given that pricing models typically correlate with token consumption. Therefore, optimizing token usage through efficient function call design and strategic planning of interactions is essential to mitigating these constraints, ensuring that the system remains functionally effective and economically viable.

At the time of writing this article, the context length for a GPT-3.5-turbo-0613 is 4,096 tokens, and for a GPT-4-turbo, it is 128,000 tokens. The small context window combined with a large number of defined functions can cause the loss of critical early context or information. Another problem that arises from this is the price since, in the worst case, the price equals the length of the context window. For example, as seen in Table 1, if we would allocate 100 tokens for the completion, the prices would be:

The following "inconvenient" part is the execution of the function call itself since two requests are sent to get the response from the GPT. The first request contains definitions of all functions and is responsible for choosing the right one regarding user questions. The second generates the response for the user based on the

Table 1: Example table for prices where the prompt_tokens = $context\_length - 100$.

Model	Context-length	Price[1]
GPT-3.5-turbo-0613	4 096	0.002148
GPT-4-turbo	128 000	1.292

[1] $price = (input\_tokens * input\_price) + (output\_tokens * output\_price)$, where for GPT-3.5-turbo-0613: $input\_price = 0.0000005\$$, $output\_price = 0.0000015\$$ and for GPT-4-turbo: $input\_price = 0.00001\$$, $output\_price = 0.00003\$$, the prices are for 1 token [2];

function's output. The entire process is illustrated in Figure 1.

1. The GPT-3.5 acts as an assistant; there are 27 functions available with the following prompt: "Don't make assumptions about what values to plug into functions. Ask for clarification if a user request is ambiguous."

2. User asks: "What is the weather today in Prague in Celsius?" The GPT sends all the functions to evaluate which one to use. Note that this happens with every request; tracking history together with many functions can significantly increase token consumption.

3. In the second request, GPT will process the function and generate a response based on the output.

4. The response is sent to the user. The total token usage in this example is 1300 tokens.

3 Background and Motivation

3.1 Embeddings

Text embeddings are vector representations of natural language that encode its semantic information. They are widely used in various natural language processing (NLP) tasks, such as information retrieval (IR), question answering, semantic textual similarity, bitext mining, item recommendation, etc. [17] In the field of IR, the first-stage retrieval often relies on text embeddings to efficiently recall a small set of candidate documents from a large-scale corpus using approximate nearest-neighbor search techniques [4, 6].

We need a distance function to measure the relatedness between two vectors (representing words). A popular distance function is cosine similarity, which measures the cosine of the angle between two vectors. For word embeddings, the vectors represent the semantic meaning of words in a high-dimensional space. Cosine similarity ranges from -1 to 1, with 1 indicating identical vectors, 0 indicating no similarity, and -1 indicating opposite vectors. The advantage of cosine similarity is that it can be computed using a dot product [18].

3.2 Vector database

A vector database is a specialized database that stores, manages, and retrieves data in high-dimensional vectors. Unlike traditional relational databases that manage structured data in tables with rows and columns, vector databases are optimized for handling unstructured data, such as text, images, and audio, which can be represented as vectors through various embedding techniques.

Vectors, in this context, are embeddings that capture the semantics and features of the data. For example, words, sentences, or documents can be converted into vectors using Word2Vec [9], GloVe [12], or transformer-based models in natural language processing. These vectors reside in a high-dimensional space where the distance between them reflects their semantic similarity. Vector databases enable efficient similarity searches, which is crucial for recommendation systems, image recognition, and semantic search applications.

The architecture of vector databases is designed to support operations like nearest neighbor search, which identifies vectors closest to a given query vector. This is typically achieved through indexing techniques like locality-sensitive hashing (LSH) [7], tree-based methods (e.g., KD-trees [13]) [16], or more advanced structures such as graph-based indices (e.g., HNSW—Hierarchical Navigable Small World graphs [8]). These indexing methods allow the database to handle large volumes of vectors and perform rapid similarity searches [11, 15].

3.3 Function calls

OpenAI function calling involves creating a set of callable functions that the AI model can recognize and use to perform specific actions or retrieve information from external systems. These functions are defined using the OpenAPI specification, which describes the endpoints, request and response structures, and other necessary details in a standardized format. This specification acts as a contract between the AI model and the external services, ensuring precise and consistent communication.

Developers define functions using the OpenAPI specification, detailing function names, input parameters, etc. This enables them to identify appropriate function calls based on user input or conversational context.

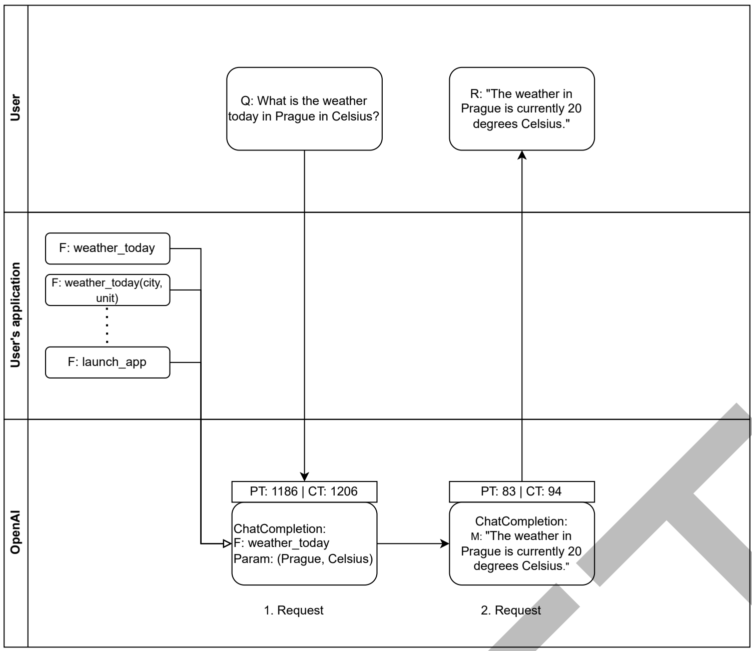

Figure 1: Default execution of the function call on the backend with the real example, where Q = Question, F: Function, PT: Prompt tokens, CT: Completion tokens, R: Response

When the AI model needs external data or actions during runtime, it generates a function call according to the specification, routing it to the corresponding external service or endpoint. The response from the external service is processed and integrated into the AI model's output, allowing it to provide enriched and contextually relevant responses by leveraging real-time data or performing specific actions. The disadvantage of the function calls is that they consume a considerable amount of tokens since the metadata of every function is sent with each request.

3.4 Motivation

We aim to create an approach where we can save the user-defined functions into the vector database and query the database to retrieve and send only the functions we need to complete the user's request. We can "scale infinitely" while reducing the overall cost. This can be highly effective when used on edge devices such as mobile phones and IoT devices, due to the limited context window on such devices.

Another benefit of this approach regards functions without argument or, as we call it, "SmartFuncCall": We can execute them on the host and use the output of the given function as input to the GPT without sending the whole function in JSON and performing the function call.

4 Methodology

Our solution efficiently stores functions with their corresponding parameters in the vector database. This approach allows the vector database to be queried before the request is sent via OpenAI API, ensuring the correct function is used (if it exists). The wrapper parses the arguments into the JSON format OpenAI needs, optimizing the selection of only the necessary functions to complete the request. These functions are then sent as a payload, significantly reducing the number of tokens sent with each request. Implementing these measures can substantially reduce the energy costs associated with large language models (LLMs) inference, which are currently remarkably high. Additionally, this can significantly decrease the carbon footprint. For instance, a ChatGPT-like application, with an estimated usage of 11 million requests per hour, generates approximately 12,800 metric tons of CO_2 emissions annually [5, 14].

The correct function is retrieved from the vector database based on the query's similarity. If more than one function with a similar context is saved inside the database, we send all of these functions to the GPT, where the model will pick the correct function based on the query.

Our solution demonstrates the ability to work with two types of functions, those are:

1. **SmartFuncCall (Function without arguments)** - these functions contain predefined logic and do not accept any arguments. They simply return a string, which will GPT use as input to generate a human-like response. The benefit of this approach is that the function can run in the background, allowing you to dynamically alter the prompt without needing to do an actual function call, which can further reduce the token cost.

```python
def mass_of_sun() -> str:
    return "The mass of sun is approximately 1,989E30 kg."
```

2. **Function with one or more arguments** - The currently supported data types are *string*, *integer*, *enum*, and *array*. The output should be the sentence of a string type utilizing these arguments. GPT will then add a human-like language to it.

```python
from pydantic import BaseModel, Field

#Define Pydantic Input
class FunctionInput(BaseModel):
    name: str = Field(..., description="Name of a planet")

#Define function
def mass_of_planet(params: FunctionInput) -> str:
    return f"Approximate mass of {params.name}."
```

The next step would be to wrap these functions with the FunctionHolder. This will extract the parameters from the function and create a JSON-like schema compatible with the OpenAI function calls. The schema will then be saved inside the vector database together with the function description. The description is used later to find the function that matches the user query.

Example 1

We will use the same scenario as seen in Figure 1 but demonstrate it with our solution, where the query returns two functions from the vector database. These functions are then sent to the GPT, which decides what function to use, as shown in Figure 2.

1. The GPT-3.5 acts as an assistant; there are 27 functions available with the following prompt: "Don't make assumptions about what values to plug into functions. Ask for clarification if a user request is ambiguous."
2. The user asks, "What is the weather today in Prague in Celsius?" Based on this question, the GPT will decide what function to use. In the previous step, where we queried the vector database, we shrank the number of functions sent to the GPT to only those needed.
3. In the second request, GPT will process the function and generate a response based on the output.
4. The response is sent to the user. The total token usage in this example is 242 tokens.

Example 2

We will again use the scenario from Figure 1 but demonstrate the functionality of SmartFuncCall, where the function is executed in the user application, and the GPT processes the output. The illustration is shown in Figure 3.

1. The GPT-3.5 acts as an assistant; there are 27 functions available with the following prompt: "You are a personal assistant, please answer me to the question {question}, extract the information from this {result} and not from anywhere else, and add some more semantics to it aka make it more human-like.". Where: The question is user input and the result is an output from the function
2. The user asks, "What is the weather today ?" Since the question has no parameters, it will be directly executed in the user's application, and the result and the question will be part of the dynamic prompt.
3. In the request, GPT will process the prompt and generate a response.
4. The response is sent to the user. The total token usage in this example is 93 tokens.

5 Results

We have tested the proposed solution on the ten questions (Table 5) with and without a vector database[2]. The benchmark measured input and completion tokens and the completion time. These ten prompts had either none or some arguments in the format of *string*, *integer*, or *array*.

As mentioned in Section 4, some functions saved in the vector database had no parameters. These functions were used to demonstrate the functionality of the SmartFuncCall.

- **VectorDB** - our solution where the functions are saved in the vector database and then executed
- **defaultFC** - approach where the default function call is used

Time consumption

Table 2: Comparison of Time Consumption between VectorDB and defaultFC

Parameter	**VectorDB**	**defaultFC**
Avg. time	2.7169	2.1804
Worst time	4.511	2.833
Difference in avg. times	± 19.75%	
Difference without #4 function	± 11.60%	

Table 2 shows that the difference between the avg. times are around 20% since the fourth question is an outlier to the data, we have calculated the difference in average times without this question, the difference is then 11.6%. In the case of vectorDB, the calculation consists not only of an OpenAI function call but also of the similarity search inside the database.

[2] ChromaDB is used in the benchmarks

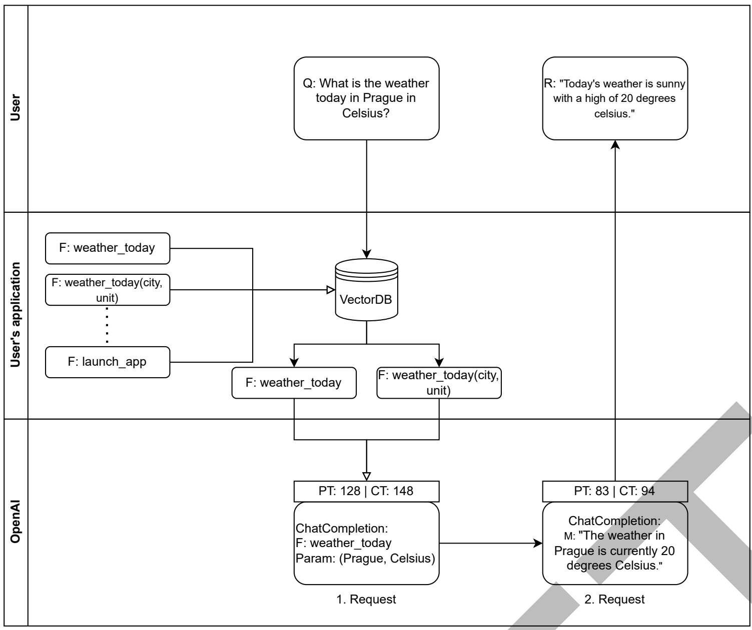

Figure 2: Execution of the function call with the vector database, where Q = Question, F: Function, PT: Prompt tokens, CT: Completion tokens, R: Response

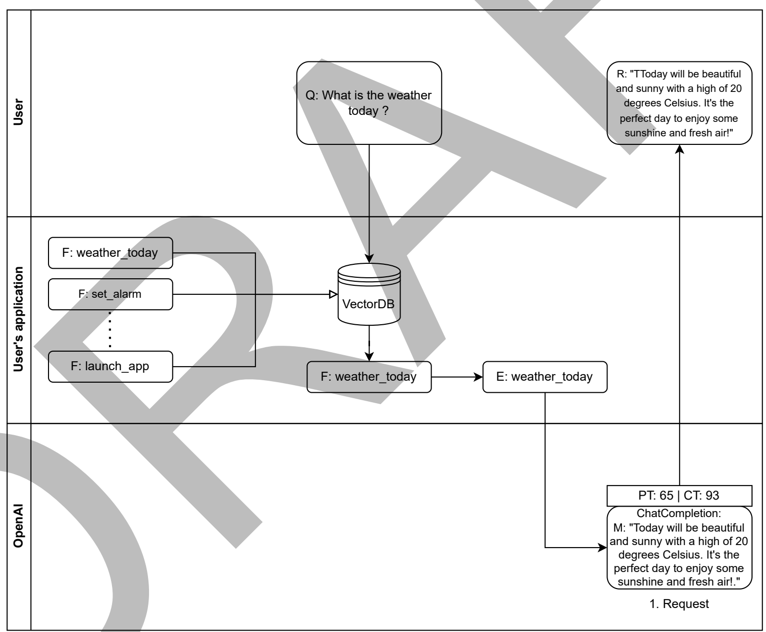

Figure 3: Execution of the SmartFuncCall with the vector database, where Q = Question, F: Function, E: Execution of a function, PT: Prompt tokens, CT: Completion tokens, R: Response

Input token consumption

The results presented in Table 3 highlight significant differences in token consumption and cost between the VectorDB and defaultFC methods. The average token consumption for VectorDB is 169.7, which is considerably lower than the 526.2 tokens consumed by defaultFC. The worst-case token consumption shows a similar trend, with VectorDB at 273 tokens and defaultFC at 585 tokens. Overall, the total token consumption is 1697 for VectorDB and 5262 for defaultFC.

This reduction in token usage translates directly to cost savings. The average price per token for VectorDB is \$0.000085, compared to \$0.0002925 for defaultFC. Consequently, the total price for using VectorDB is \$0.000849, whereas defaultFC costs \$0.002626. The differences in average consumption and average price are approximately 210.1% and 244.1%, respectively, indicating substantial savings when using VectorDB.

Table 3: Comparison of Input Token Consumption and Pricing between VectorDB and defaultFC

Parameter	VectorDB	defaultFC
Avg. token consumption	169.7	526.2
Worst token consumption	273	585
Total tokens	1697	5262
Difference in avg. consumption	± 210.1%	
Avg. price ($)	0.000085	0.0002925
Total price ($)	0.000849	0.002626
Difference in avg. price	± 244.1%	

Table 4: Comparison of Total Token Consumption and Pricing between VectorDB and defaultFC

Parameter	VectorDB	defaultFC
Avg. token consumption	205.2	563.6
Worst token consumption	321	649
Total tokens	2052	5636
Difference in avg. consumption	± 174.7%	
Avg. price ($)	0.0001383	0.0003486
Total price ($)	0.001382	0.003187
Difference in avg. price	± 152%	

Total token consumption

Table 4 compares token consumption and pricing between our approach (VectorDB) and default function calling from OpenAI (defaultFC). VectorDB's average token consumption is 205.2 tokens, significantly less than defaultFC's 563.6 tokens. In worst-case scenarios, VectorDB uses 321 tokens compared to 649 tokens for defaultFC, demonstrating greater efficiency. Overall, VectorDB consumes 2052 tokens, while defaultFC consumes 5636 tokens, indicating a substantial efficiency advantage for VectorDB. The difference in average consumption is ±174.7%. Pricing also favors VectorDB, with an average token cost of $0.0001383 versus $0.0003486 for defaultFC.

Total costs are $0.001382 for VectorDB and $0.003187 for defaultFC, reflecting a ±152% difference in average price, making VectorDB more cost-effective.

6 Conclusion

As proposed in this paper, the implementation of selective function calls within the OpenAI API ecosystem addresses a critical efficiency challenge.

- **Token Reduction:** Our solution significantly reduces the number of tokens consumed per request by leveraging a vector database to store and retrieve function definitions based on similarity scores. This method shows a remarkable decrease in average prompt token consumption and a reduction in prompt input costs compared to the default function call mechanism.

- **Compatibility with Any LLM Supporting Function Calling**: Our approach offers modularity for the vector database and the LLM. The tested ChromaDB can be swapped with another vector database, and the solution is compatible with other LLM models that support function calls.

- **Energy Saving and Reduced Carbon Footprint:** The potential of our solution is to enhance the scalability and affordability of utilizing GPT's capabilities and lower the overall energy cost, which in turn reduces the carbon footprint.

- **Functionality on Small Devices:** This adaptability is particularly effective for edge devices like mobile phones and IoT devices. Given the limited context window on such devices, our proposed solution can maximize functionality.

The code with the implementation and benchmark is available at: https://github.com/Rankacy/CLaRa

Acknowledgement: We would like to express my sincere gratitude to Rankacy for their unwavering support and provision of resources throughout the course of this research.

References

[1] Function calling and other api updates, available at: https://openai.com/index/function-calling-and-other-api-updates/, 2023.

[2] Pricing, available at: https://openai.com/api/pricing/, 2024.

[3] Brown, T. B., et al. Language models are few-shot learners, 2020.

[4] Cao, H. Recent advances in text embedding: A comprehensive review of top-performing methods on the mteb benchmark, 2024.

[5] Chien, A. A., Lin, L., Nguyen, H., Rao, V., Sharma, T., and Wijayawardana, R. Reducing the carbon impact of generative ai inference (today and in 2035). In *Proceedings of the 2nd Workshop on Sustainable Computer Systems* (2023), pp. 1–7.

[6] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. Indexing by latent semantic analysis. *Journal of the American society for information science 41*, 6 (1990), 391–407.

[7] Gionis, A., Indyk, P., and Motwani, R. Similarity search in high dimensions via hashing. *Proceeding VLDB '99 Proceedings of the 25th International Conference on Very Large Data Bases 99* (05 2000).

Table 5: Questions used in benchmarks

Question	Parameters
1. Launch a Spotify app please	None
2. Can you please play Walk n Skank by Macky Gee on Spotify	None
3. What is the current time and date?	None
4. What is the weather today?	None
5. What is the weather today in San Francisco in Celsius?	city: San Francisco, unit: Celsius
6. Can you please create a new task: feed the dog with high priority for 15:00 today	text: feed the dog, date_time: 1635896400, priority: High
7. Can you please dial 420696669	number: 420696669
8. Can you play Let Me Hold You by Netsky on Spotify please	song_name: Let me hold you by Netsky
9. Can you translate hello, how are you from English to Spanish	text: Hello, how are you, from_language: English, to_language: Spanish
10. I'm going to Zabka and I need Apples, milk, beer, coffee, bread, and ketchup	store: Zabka, items: ['Apples', 'milk', 'beer', 'coffee', 'bread', 'ketchup']

Table 6: Answers and information about whether the function call was required or executed on the host without sending it to the GPT. The questions are specified in Table 5.

Q.	Answer	Function Call
1.	Sure thing! Let me go ahead and launch the Spotify app for you. Just give me a moment to get everything up and running. Opening Spotify now! Enjoy your music. Let me know if you need any help navigating the app.	No
2.	Sure thing! I've got Walk n Skank by Macky Gee queued up and ready to play for you on Spotify. Get ready to groove to this awesome track! Enjoy!	No
3.	The current time is 1:24 pm, and today's date is March 25th, 2024. How can I assist you further?	No
4.	Today will be a beautiful day with clear skies and sunshine, perfect for enjoying the outdoors. The temperature will reach a pleasant high of 20 degrees Celsius. Enjoy the weather!	Yes (2 functions had close similarity, the one with no args is used)
5.	The weather in San Francisco is currently 20 degrees Celsius.	Yes (2 functions had close similarity, the one with args is used)
6.	Task "Feed the dog" has been created with high priority for 15:00 today.	Yes
7.	I have dialed the number 420696669.	Yes
8.	The song "Let me hold you" by Netsky is now playing on Spotify. Enjoy!	Yes
9.	Hola, ¿cómo estás?	Yes
10.	Your grocery list for Zabka includes: Apples, milk, beer, coffee, bread, and ketchup.	Yes

[8] MALKOV, Y. A., AND YASHUNIN, D. A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence 42*, 4 (2018), 824–836.

[9] MIKOLOV, T., CHEN, K., CORRADO, G., AND DEAN, J. Efficient estimation of word representations in vector space, 2013.

[10] OPENAI. Gpt-4 technical report, 2023.

[11] PAN, J. J., WANG, J., AND LI, G. Vector database management techniques and systems. In *SIGMOD Conference Companion* (2024), pp. 597–604.

[12] PENNINGTON, J., SOCHER, R., AND MANNING, C. Glove: Global vectors for word representation. vol. 14, pp. 1532–1543.

[13] PROCOPIUC, O., AGARWAL, P. K., ARGE, L., AND VITTER, J. S. Bkd-tree: A dynamic scalable kd-tree. In *Advances in Spatial and Temporal Databases: 8th International Symposium, SSTD 2003, Santorini Island, Greece, July 2003. Proceedings 8* (2003), Springer, pp. 46–65.

[14] SAMSI, S., ZHAO, D., MCDONALD, J., LI, B., MICHALEAS, A., JONES, M., BERGERON, W., KEPNER, J., TIWARI, D., AND GADEPALLY, V. From words to watts: Benchmarking the energy costs of large language model inference. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)* (2023), IEEE, pp. 1–9.

[15] STATA, R., BHARAT, K., AND MAGHOUL, F. The term vector database: fast access to index-

ing terms for web pages. *Computer Networks 33*, 1-6 (2000), 247–255.

[16] Venables, W. N., Ripley, B. D., Venables, W., and Ripley, B. Tree-based methods. *Modern applied statistics with S-Plus* (1999), 303–327.

[17] Wang, L., Yang, N., Huang, X., Yang, L., Majumder, R., and Wei, F. Improving text embeddings with large language models, 2024.

[18] Xia, P., Zhang, L., and Li, F. Learning similarity with cosine similarity ensemble. *Information sciences 307* (2015), 39–52.