EpiGEN: An Efficient Multi-Api Code GENeration Framework under Enterprise Scenario

Sijie Li[1,*], Sha Li [2,*], Hao Zhang[2], Shuyang Li[2], Kai Chen[2], Jianyong Yuan[2]
Yi Cao[2,✉], Lvqing Yang[1,✉]
[1]School of Informatics, Xiamen University, [2]Huawei Technologies
[1]Xiamen, China, [2]Shenzhen, China
[1]30920211154161@stu.xmu.edu.cn
[1]lqyang@xmu.edu.cn
[2]{lisha45, zhanghao339, lishuyang, colin.chenkai, yuanjianyong, caoyi16}@huawei.com

Abstract

In recent years, Large Language Models (LLMs) have demonstrated exceptional performance in code-generation tasks. However, under enterprise scenarios where private APIs are pre-built, general LLMs often fail to meet expectations. Existing approaches are confronted with drawbacks of high resource consumption and inadequate handling of multi-API tasks. To address these challenges, we propose EpiGEN, an **E**fficient multi-A**pi** code **GEN**eration framework under enterprise scenario. It consists of three core modules: Task Decomposition Module (TDM), API Retrieval Module (ARM), and Code Generation Module (CGM), in which Langchain played an important role. Through a series of experiments, EpiGEN shows good acceptability and readability, compared to fully fine-tuned LLM with a larger number of parameters. Particularly, in medium and hard level tasks, the performance of EpiGEN on a single-GPU machine even surpasses that of a fully fine-tuned LLM that requires multi-GPU configuration. Generally, EpiGEN is model-size agnostic, facilitating a balance between the performance of code generation and computational requirements.

Keywords: LLM, Code Generation, Private-Library, LangChain

1. Introduction

Code-generation LLMs such as Codex (Chen et al., 2021), CodeGen (Nijkamp et al., 2022), and CodeGeex (Zheng et al., 2023) have been trained on a vast amount of open-source code. Their sophisticated coding capabilities, comparable to those of skilled programmers, have not only revolutionized traditional software engineering practices but also significantly improved development efficiency. Nonetheless, in enterprise environments, private API libraries and coding standards are usually established to ensure code security and reusability. While training data of LLMs are primarily sourced from open platforms like GitHub and Stack Overflow, private libraries are rarely included. As a result, LLMs cannot effectively call private APIs to perform code generation tasks under enterprise scenarios.

(Zan et al., 2023) introduce an innovative solution that divides the code generation task into two modules: APIFinder and APICoder. It first performs vector search to find the corresponding APIs, and then uses a large amount of private code to incrementally pre-train a LLM for code generation. Although this approach has its unique advantages, its reliance on resources and data hinder its adoption under enterprise-level scenarios. Furthermore, when it comes to more than two API-tasks, the effectiveness of the generated code decreases significantly. Another prevalent strategy involves collecting a substantial corpus of text pairs consisting of requirements and corresponding code snippets. LLMs with a large number of parameters are then fully fine-tuned to enable the model to gain knowledge of when and how to utilize private APIs. However, this method is highly dependent on model complexity and GPU resources, making it challenging to deploy in practice.

In enterprise environments, complex tasks are first broken down into subtasks, then assigned to multiple job roles to accomplish. Inspired by such collaborative approaches in enterprises, we aim to resolve the issues in previous methodologies by introducing EpiGEN, an **E**fficient multi-A**pi** code **GEN**eration framework suited for enterprise scenarios. EpiGEN comprises three key modules: Task Decomposition Module, API Retrieval Module, and Code Generation Module. Practical requirements are often concise but involve multiple APIs. Therefore, it is crucial to decompose the requirements into fine-grained subtasks. Furthermore, we establish an API vector library index to make the full use of enterprise's API documentation and then utilize LangChain (langchain ai,

Work done during internship at Huawei Technologies Co., Ltd.
* These authors contributed equally to this work.
✉ Corresponding author.

LREC-COLING 2024, pages 6206–6215
20-25 May, 2024. © 2024 ELRA Language Resource Association: CC BY-NC 4.0

2022) for API retrieval. Lastly, we generate high-quality code integrating requirements, subtasks, and API information.

A series of experiments were conducted to evaluate the performance of each module and validate code quality of the overall framework. The experimental results demonstrate that EpiGEN excels in task decomposition and code generation across various models, even with constraints of limited resources and data. Our contributions can be summarized as follows:

- To the best of our knowledge, EpiGEN is the first enterprise-oriented, highly adaptable code generation framework.
- EpiGEN effectively handles complex tasks that involve multiple APIs, even with limited resources and data.
- Extensive experiments have verified excellent performance of EpiGEN on invoking private APIs and code generation.

2. Related Work

2.1. Pre-training code model

Codex (Chen et al., 2021), powered by large language pre-trained models, has gained robust code generation capabilities through extensive and high-quality code corpus training. This achievement has inspired emergence of models like PaLM-Coder (Chowdhery et al., 2022), PanGu-Coder (Christopoulou et al., 2022; Shen et al., 2023), and AlphaCode (Li et al., 2022), all of which boast large-scale parameters and have demonstrated outstanding performance in specific code-related tasks. Regrettably, none of these models are open-sourced. Currently, there have been remarkable open-source models introduced into the field, such as CodeParrot (Huggingface, 2021), CodeGen (Zan et al., 2023), and PolyCoder (Xu et al., 2022). These models have greatly contributed to the advancement of code generation. However, they remain constrained by the architecture of generative models, limiting them to left-to-right code generation. In response to this limitation, models like SantaCoder (Allal et al., 2023), StarCoder (Li et al., 2023), and WizardCoder (Luo et al., 2023) have emerged to support code generation at arbitrary positions within the code. Additionally, code generation models are enhancing their versatility, with models like CodeGeeX (Zheng et al., 2023), BLOOM (Scao et al., 2022), and ERNIE-Code (Chai et al., 2022) designed to meet multi-lingual programming demands. DocCoder and APIcoder (Zan et al., 2023) aim to empower language models with the ability to call APIs, addressing the requirements of programmers writing code with private libraries. In recent times, code-llama (Rozière et al., 2023) achieved notable results by fine-tuning llama2 (Touvron et al., 2023) on meticulously curated high-quality guided code datasets.

2.2. Dense Retrieval

Vector retrieval (Berry et al., 1999) is primarily based on comparing dense vectors and often involves the use of approximate nearest neighbor (ANN) techniques, which relax the requirement for exact search by allowing a small number of errors to enhance retrieval efficiency. Among the popular ANN methods, such as tree algorithms (Muja and Lowe, 2014; Houle and Nett, 2014), locality-sensitive hashing (Andoni et al., 2015; Indyk and Motwani, 1998), product quantization (Wang et al., 2013; Norouzi et al., 2013), Proximity graph (Arya and Mount, 1993), and HNSW (Malkov and Yashunin, 2018), the index based on Hierarchical Navigable Small World networks (HNSW) is widely recognized as the current state-of-the-art. The Faiss library (Johnson et al., 2019) offers a widely adopted and implementation of the HNSW index, which become a standard baseline. Although conceptually similar (Lin, 2022), it is evident that top-k retrieval for sparse vectors and dense vectors require distinct software stacks. Since it has been demonstrated that hybrid approaches of dense and sparse representations are more effective (Ma et al., 2022; Lin and Lin, 2023), many modern systems integrate separate retrieval components to achieve mixed retrieval. For example, the Pyserini Information Retrieval (IR) toolkit (Lin and Lin, 2023) incorporates both sparse retrieval and dense retrieval, leveraging Lucene (Lin et al., 2023) and Faiss, respectively.

3. Preliminaries

3.1. Task Definition

Given a requirement, the objective of EpiGEN is to generate code that fulfills the requirement by utilizing APIs from the private library. Figure 1 illustrates an example of the task's input and output. Specifically, the input involves a natural language description of a problem that the end-user aims to implement, with the corresponding target code generated as output. In EpiGEN, the processing steps and API library remain invisible to the user.

3.2. Scenario description

This paper will introduce EpiGEN in detail using the W scenario of company H as an example. In order to improve code standards and reusability,

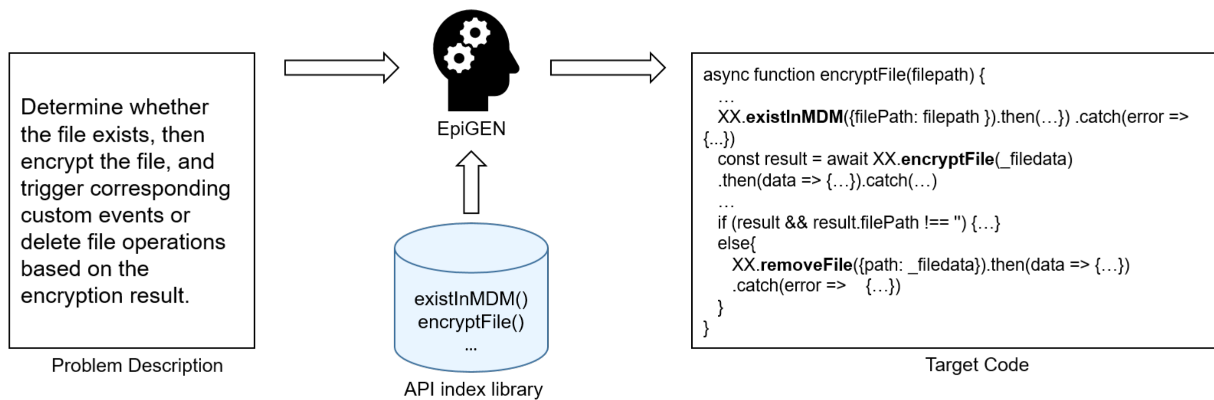


Figure 1: A typical case of using EpiGEN to solve a problem. The input is a string of problem description, which requires invoking three private APIs, and the output is source code that addresses the requirement.

company H has constructed a JavaScript API library in the W scenario that includes 97 APIs, covering various practical requirements.

3.3. Vector retrieval with LangChain

LangChain is a framework designed to enhance application functionality by leveraging LLMs. Its primary objective is to enable developers to utilize LLMs more easily and interact with various data sources and applications. It also provides various encapsulated classes to assist users in quickly implementing vector retrieval.

- Document loader: This component loads files into a list of documents and performs vectorization in a unified manner. It supports CSV, PDF, and JSON data formats.
- Embeddings: This feature represents documents uniformly.
- Vectorstore: This component stores vectorized documents and provides various methods for calculating similarity in vector retrieval.

3.4. API Document Definition

Common API documents contain the following components:

- API Name: Unique identifier for the API that represents basic information.
- API Functionality: Detailed explanation of the functionality that the API can achieve.
- API Parameter: Parameters required by the API, including expected data types and formats.
- API Examples: Examples of how to use the API.

In the private API library of scenario W, each API is comprised of two components: API Des and API Exp. 'API Des' includes API Functionality and API Parameter, and 'API Exp' comprises API Name and API Examples. Moreover, we define API All, which encompasses all the components of an API.

4. Methods

The workflow of the EpiGEN, as depicted in Figure 2, consists of three main components: the Task Decomposition Module (TDM), the API Retrieval Module (ARM), and the Code Generation Module (CGM). The meanings represented by different color blocks are illustrated in Figure 3.

4.1. Task Decomposition Module

Task Decomposition Module (TDM) is composed of an efficiently fine-tuned LLM. When TDM receives a requirement, its primary task is to understand the requirement and then decompose it into several fine-grained subtasks that can be implemented by private APIs. Utilizing a foundational model to directly execute this task may lead to subtasks that have low similarity to the APIs in the private library. Therefore, we have chosen an efficient fine-tuning strategy to effectively inject knowledge of W scenario into the large model.

4.1.1. Training Corpus

Our initial corpus contains only 273 samples, as shown in Figure 4. This is insufficient to train a task decomposition model that meets our expectations. This challenge often occurs in enterprise where high-quality labeled data is lacking. To enhance the model's understanding of user requirements, we design a data reconstruction and data augmentation strategy.

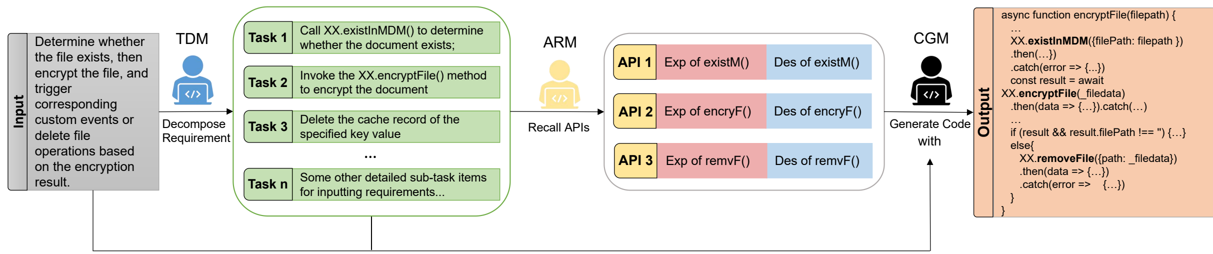


Figure 2: The overview of EpiGEN. It consists of three modules: TDM: decompose the requirements into fine-grained subtasks. ARM: recall APIs from the API library based on subtasks. CGM: generate code based on the result of TDM, ARM and initial requirements.

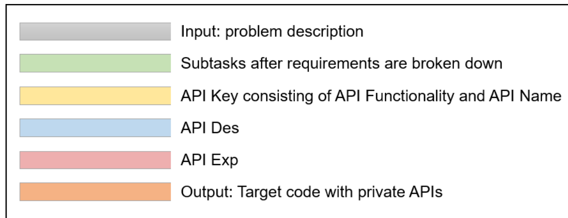


Figure 3: The meanings represented by different colored blocks.

<table>
<tr><th>Code</th><th>APIs</th></tr>
<tr><td>export const handleGetTenantId = async () =>
{
const _net = await XX.getNetworkType()
if (!_net || !_net.status) {
XX.showToast ({
msg: i18n.t('email: network connection failed. Please check the network.') , type:'n'.})
...
}</td><td>1. XX.getNetworkType();
2. XX.showToast({ msg: 'email:network connection failed. Please check the network', type: 'n' })
3. XX.fetch(url, { method: 'get', headers: { timeout: 6000 } }).then((res) => res.json()).then((reply) => { return reply}).catch((error) => { XX.hideLoading() throw Error(error)})</td></tr>
</table>

Figure 4: An example of original corpus.

Requirement	Structured Steps
Check the network connection, initiate a network request, and obtain the TenantId result.	1. Check the network connection. If the network connection fails, use the XXX.showToast() interface to display the prompt information. 2. Call getNetworkType to obtain the current network status. 3. If the network connection fails, a message is displayed and recorded in logs. 4. Use the XXX.fetch() interface to send a fetch request and transfer the interface URL and relative path. 5. Set the timeout period to 6000 ms. 6. Parse the response and check whether the response is correct. 7. If TenantId is obtained successfully, the result is returned. 8. If an exception occurs, the error log is recorded and null is returned.

Figure 5: An example of training corpus of TDM.

For each line of corpus, we first query the API Document to determine the functionalities of each API used. Then a prompt formatted with above information, combined with the code, was submitted to the LLM. Our aim is to leverage the LLM's capabilities to reveal the requirements accomplished by the given code. Secondly, in order to obtain the sub-tasks required to solve the problem, we design an additional prompt. This involves combining the requirements from the first step, used APIs, and code into a prompt and querying the LLM again. Finally, we obtain the training corpus shown in Figure 5, with the results of the first two steps serving as the input and output for our training corpus. Although 273 data points cover all private APIs under W scenario, the dataset is still too limited in size to effectively train a model. Therefore, we implement data augmentation by leveraging LLM. This strategy involves providing LLMs with descriptions of requirements and instructing them to generate multiple variations of these descriptions. The objective is to maintain the fundamental essence of each requirement while presenting it in different forms. As a result, we obtain a task decomposition corpus with a size of (2203, 2).

4.1.2. Task Decomposition Model

The TDM is responsible for breaking down user requirements, so it must have strong capabilities of contextual understanding. ChatGLM2-6B, with its excellent model architecture based on the autoregressive masked language model GLM (Zeng et al., 2022; Du et al., 2022), has an advantage in NLU tasks and demonstrates low resource consumption with an inference memory usage of 13GB. Its excellent multilingual understanding of Chinese and English make it our choice as a baseline for the EpiGEN's Task Decomposition Module. For efficient fine-tuning, we employ P-Tuning v2, showing significant effectiveness among small-scale models with fewer than 10 billion parameters.

4.2. API Retrieval Module

The API retrieval module, leveraging the capabilities of LangChain, calculates similarity coefficients from the vector database for each decomposed subtask sequentially. When the cosine distance falls below the threshold, it retrieves the corresponding API Des and API Exp from Redis, as illustrated in the Figure 6.

4.2.1. Construction of API Index

If we directly use API All as the index for the vector database, we may encounter the following challenges:

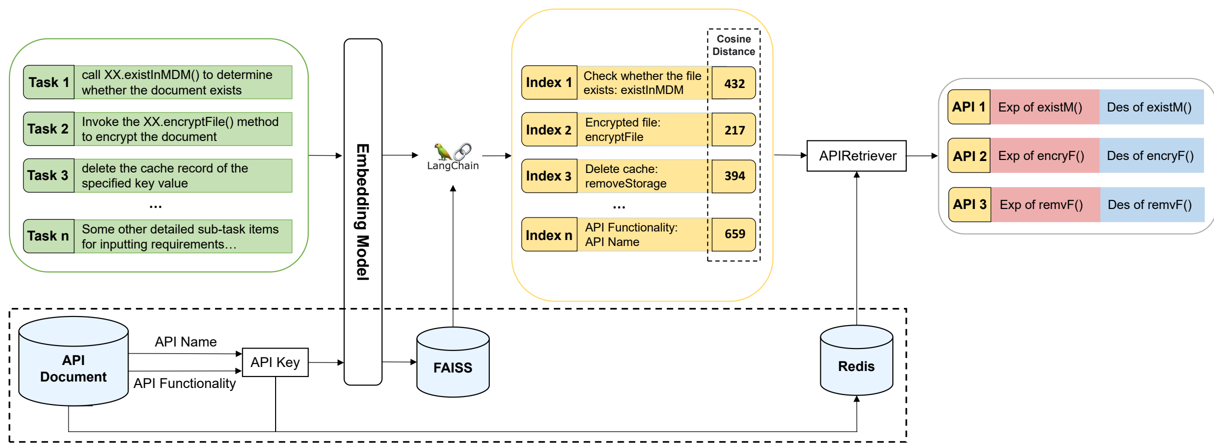
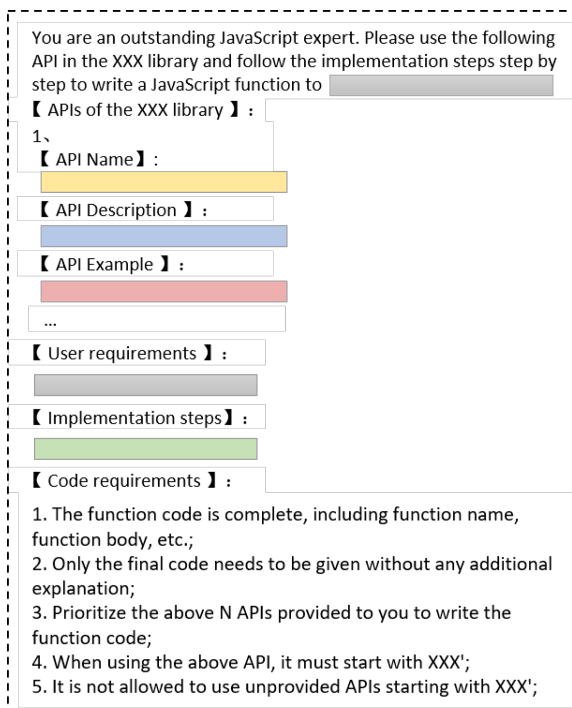


Figure 6: Workflow of the ARM. The objective of APIRetriever is to query the corresponding API Des and API Exp by API Key in Redis.



You are an outstanding JavaScript expert. Please use the following API in the XXX library and follow the implementation steps step by step to write a JavaScript function to

【 APIs of the XXX library 】：

1、

【 API Name】：

【 API Description 】：

【 API Example 】：

...

【 User requirements 】：

【 Implementation steps】：

【 Code requirements 】：

1. The function code is complete, including function name, function body, etc.;
2. Only the final code needs to be given without any additional explanation;
3. Prioritize the above N APIs provided to you to write the function code;
4. When using the above API, it must start with XXX';
5. It is not allowed to use unprovided APIs starting with XXX';

Figure 7: Example of a prompt.

- Current open-sourced vector models do not effectively represent code.
- Different APIs may possess identical parameter or descriptions, potentially causing unnecessary interference in similarity calculations.

To address these issues, we extract the most detailed and easily retrievable information from each API, focusing specifically on API Functionality and API Name (as mentioned in Section 3.4), ensuring no inclusion of code, and then store them in FAISS after embedding processing. To extract API Functionality from API Des and API Name from API Exp, we designed the following two regular expression rules: 1) Extract the API Functionality before the first message terminator in API Des; 2) Extract the API Name located between H(prefix of the private API library) and '()' in API Exp. After combining the extracted API Name and API Functionality with ": ", we define it as the API Key. In order to preserve all information of the API, we store the corresponding API Des and API Exp as the value associated with the API Key in the Redis.

4.2.2. Fine-Tuning of Embedding model

Considering the specific requirements of W scenarios and custom API coding specifications, we experiment with a fine-tuned embedding model. In this scenario, the amount of available labeled data does not meet the requirements for supervised training. Therefore, we adopt TSDAE (Wang et al., 2021) to perform unsupervised fine-tuning on the vector model. During the training process, the encoder and pooling layers first encode noisy sentences into fixed-size vectors and require the decoder to reconstruct the original sentences from these sentence representations.

4.3. Code Generation Module

We generate code by combining user requirements, subtasks, corresponding API descriptions, and API examples into a single prompt (as shown in Figure 7) and query it with LLM. Following the Chain of Thought approach, we guide the model to generate the desired code step by step according to the provided instructions. In this module, we employ ChatGLM2-6B as the code generation model.

5. Experiments

In this section, a comprehensive series of experiments is designed to evaluate the EpiGEN. Specif-

ically, we start by discussing computational resources and parameter settings, and then provide a detailed overview of some experiments from the perspectives of objectives, details, and results. Finally, we conduct an intensive analysis of our experimental findings.

5.1. Experimental Setup

5.1.1. Experimental Details

All of the training and inference tasks of our framework are conducted on a single NVIDIA V100 32GB GPU.

For the task decomposition model, we use a training-validation split ratio of 4:1 and a batch size of 1, train for 1500 steps, select a prefix length of 128, and the total training time is approximately 10 hours.

Regarding the vector model, we use an unsupervised fine-tuning approach with a batch size of 16 and a learning rate set to 3e-5, training it for 50 epochs, and the total training time is 45 minutes.

While developing the EpiGEN, we conduct full-parameter fine-tuning on PanGu-38B model using a combination of original W scenario corpus and 20,000 pieces of open-source data. The model is trained for 10 hours using 8 Ascend GPUs. The fine-tuned model is named PanGu-38B-FT and is tested in conjunction with EpiGEN.

5.1.2. Test set

To simulate actual task requirements, we ask domain experts of the private library to manually create a test set with a size of (310, 2), based on their daily job tasks in an enterprise environment, which is covering all APIs in the private library. This set is subsequently divided into two parts. The first part is the API-only task, which requires correctly providing the APIs needed to address the requirement. The second part is the Content task, which requires providing the complete function code involving private APIs. Both parts of the test set are divided into three levels of difficulty. Level 1 consists of solely one API, with a simple and straightforward requirement. Level 2 encompasses 1-2 APIs, with an equally apparent requirement. Level 3 involves two or more APIs, and the requirement description becomes more obscure.

5.1.3. Evaluation Metrics

For the task decomposition module, we use BLEU to measure similarity between the framework's output and the reference subtask items, and ROUGE to evaluate whether the framework captures the information in the original task. Both are common metrics in natural language processing, with the former tending to measure the accuracy and precise matching degree of task decomposition, similar to Precision, and the latter focusing on measuring the completeness of the decomposed tasks, similar to Recall. For the API retrieval module, we use API recall rate to measure its performance. For the code generation module, we rely on domain experts to score from three aspects: API-Integrity, Content-readability, and Content-acceptability. API-Integrity evaluates the completeness of APIs provided for API-only tasks. Content-readability focuses on the implementation logic and code style of the generated code for Content tasks. Content-acceptability mainly focuses on the completeness of function implementation, API usage, and the correctness of API parameters and other issues related to actual operation.

5.2. Main Result

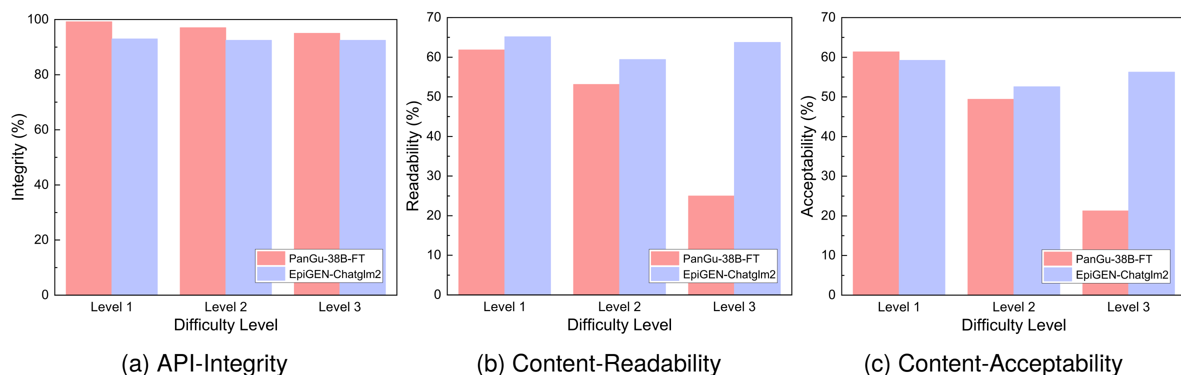
5.2.1. Performance of EpiGEN

The performance of EpiGEN and PanGu-38B-FT on the test set is shown in Figure 8. The results indicate the following:

For the API-only tasks. PanGu-38B-FT demonstrates strong API provision capabilities, accurately providing the private APIs based on requirements. Despite limited resources and training data, EpiGEN also demonstrates impressive performance in this section, with its API provision metrics comparable to those of PanGu-38B-FT. The performance of both approaches remains unaffected by different task difficulties.

For the Content tasks. PanGu-38B-FT performs well at level 1. However, both readability and acceptability tend to decrease as task difficulty increases. In contrast, EpiGEN maintains relatively consistent performance across various difficulty levels, achieving a content-readability of 63.75% and a content-acceptability of 56.25% even in the complex tasks of level 3. This can be attributed to our task decomposition module, which demonstrates robust generalization abilities following fine-tuning on a wide range of high-quality data. Complex tasks are comprehensively comprehended and subsequently decompose them into subtasks that can be implemented using the private APIs.

We can draw a valid conclusion from our experiments: in terms of private API invocation capabilities, EpiGEN is comparable to models with a larger number of parameters; in terms of complex tasks, EpiGEN shows better performance. Moreover, EpiGEN consumes much lower resources and less data.



(a) API-Integrity (b) Content-Readability (c) Content-Acceptability

Figure 8: Performance of EpiGEN and PanGu-38B-FT on different tasks

TDM	CGM	API-Integrity(%)			Content-Readability(%)			Content-Acceptability(%)		
		level 1	level 2	level 3	level 1	level 2	level 3	level 1	level 2	level 3
None	GLM	78.50	23.11	15.75	57.34	10.50	3.82	55.75	9.87	2.00
GLM	GLM	93.00	93.46	92.50	65.15	59.42	63.75	59.25	52.57	56.25
BChuan	BChuan	93.15	93.75	93.00	67.28	59.70	64.36	61.45	55.72	56.36
Llama	Llama	95.20	94.08	93.50	69.06	61.21	60.10	65.41	57.95	58.55
PanGu	PanGu	**97.50**	**97.28**	**96.75**	**75.70**	**70.82**	**71.63**	**72.52**	**69.35**	**68.85**

Table 1: The result of different models in different levels, difficulty increases from level 1 to 3. GLM represents ChatGLM2-6B, BChuan represents BaiChuan2-7B, Llama represents Llama2-7B, and PanGu represents PanGu-38B.

5.2.2. Generalization of EpiGEN

To demonstrate the strong generalization capabilities of our framework, we select ChatGLM2-6B, BaiChuan2-7B (Baichuan, 2023), Llama2-7B, and PanGu-38B as task decomposition models or code generation models and evaluate their performance in pairs. The results are displayed in Table 1. Experimental results indicate that EpiGEN performs impressively across various task decomposition models and code generation models. In API-only tasks of different difficulty levels, EpiGEN consistently achieves a minimum API-Integrity score of 93.00%. Similarly, in content-only tasks with increasing difficulty, EpiGEN maintains stable performance without a sharp decline as the number of required APIs increases. Moreover, as the model's parameter size increases from 6B to 7B, and ultimately to 38B, EpiGEN's performance follows an upward trend. Therefore, it is inferred that, given sufficient resources, the capabilities of EpiGEN would improve along with the enhancement of the model's capacity.

5.3. Intensive Analysis

We are curious about the effectiveness of certain components and tricks in EpiGEN. Therefore, we design some experiments to address the following questions.

Q1: Is the task decomposition module essential? We design experiments to compare the performance with and without the task decomposition module. The results, presented in Table 1, show a substantial decrease in performance metrics, both in the API-only and Content tasks across all levels, when the framework operates without the TDM. It concludes that the Task Decomposition Module (TDM) significantly improves EpiGEN's capability in private API invocation. Furthermore, TDM plays a critical role in effectively handling multi-API tasks.

Q2: How does the TDM perform in terms of adaptability? We efficiently fine-tuned BaiChuan2-7B, Llama2-7B, and PanGu-38B as our task decomposition models. BLEU and ROUGE are employed as evaluation metrics, as shown in Table 2. All four models achieve a BLEU-4 score of 51% and an ROUGE-L score of 60%. Notably, PanGu-38B performs the best as the LLM with largest number of parameters. This design pattern is observed to exhibit strong adaptability because the requirements are fulfilled using smaller-parameter models.

Model	BLEU-4	ROUGE-L
PanGu-38B	58.2543	69.2712
ChatGLM2-6B	51.2987	60.2304
BaiChuan2-7B	53.8426	62.9987
Llama2-7B	55.7658	65.3381

Table 2: Performance of TDM with different models.

Q3: Which information from the API documentation is most suitable as a vector index? Based on the analysis in the previous section, the combination of API Name and API Functionality is employed as the index for storing. Alternative indexing schemes, such as API All, API Des, and API Exp, are also explored. Experiments are designed using the API recall rate as the evaluation metric, and the results are shown in Table 3. The experimental results align with the hypothesis that current open-source embedding models lack the ability to effectively represent code vectors. Therefore, it is necessary to extract the most critical and concise information from the API document as an index to ensure a balance between representation quality and retrieval accuracy.

API Index	Recall(%)
API All	36.85
API Des	42.39
API Exp	23.61
API Key	**70.68**

Table 3: Results of different API index construction methods.

Q4: How to select open-source vector models? In ARM, unsupervised fine-tuning of open-source vector models is conducted, and the choice of the base model is crucial. The selection criteria are as follows: For API indices containing both Chinese and English, we aim to choose a multi-lingual open-source vector model with strong contextual understanding abilities. Multiple open-source models are considered for experimentation. The experimental results, as presented in Table 4, lead to the selection of m3e-base as the embedding base model. Due to its inclusion of mixed Chinese and English corpora during pretraining and the use of ReBerta as the base model, as it exhibits excellent Chinese and English representation capabilities.

Embedding Model	Recall(%)
text2vec-base-multilingual	64.51
text2vec-base-zh	47.67
text2vec-large-chinese	56.12
m3e-base	70.68
m3e-base-finetune	**76.47**

Table 4: Performance of different embedding models.

6. Discussion and Limitations

In this section, some limitations of EpiGEN are discussed: i) In CGM, all the detailed information of the required APIs is provided to the code generation model, to allow the LLMs to invoke them autonomously. However, it is observed in the experiments that this approach does not perform well in handling details such as parameter assignments. ii) It is found that when there is a certain dependency relationship between APIs, such as the output of the previous API being used as a parameter of the next API, the results tend to deviate. This issue might be alleviated as the LLM capabilities improve or larger parameters.

7. Conclusion

In this paper, EpiGEN is introduced as a resource-efficient code generation framework designed for enterprise private APIs. EpiGEN excels at handling complex tasks involving multiple API calls. To achieve these functionalities, the entire framework is decomposed into three modules: the Task Decomposition Module, API Retrieval Module, and Code Generation Module. The Task Decomposition Module is responsible for breaking down requirements into fine-grained subtasks that can be accomplished by APIs in the private library. The API Retrieval Module leverages LangChain's capabilities and employs vector retrieval to fetch the APIs corresponding to each subtask. Finally, the Code Generation Module uses prompt engineering to guide the LLM in generating code that meets the requirements. Experiments show that, even with low resource consumption and limited training data, EpiGEN performs on par with fully fine-tuned large number of parameter LLMs and excels in multi-API tasks. Furthermore, we design extensive experiments to demonstrate the generalization of EpiGEN, proving its effectiveness across different large language models.

8. Acknowledgement

The authors thank Yifan Zhu, Rongjun Xu, Jun Wang for their great support to this work. The authors also thank Chaochao Li, Liu Wu, Hairong Xia, Xue Han, Zhongqi Liao, Yachao Li, Tengyun Wang, Yuxiang Zhou, Yunliang Shi, Qiang Chen and Wenxi Li for generous help on this work.

9. Bibliographical References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal lsh for angular distance. *Advances in neural information processing systems*, 28.

Sunil Arya and David M Mount. 1993. Approximate nearest neighbor queries in fixed dimensions. In *SODA*, volume 93, pages 271–280. Citeseer.

Baichuan. 2023. Baichuan 2: Open large-scale language models. *arXiv preprint arXiv:2309.10305*.

Michael W Berry, Zlatko Drmac, and Elizabeth R Jessup. 1999. Matrices, vector spaces, and information retrieval. *SIAM review*, 41(2):335–362.

Yekun Chai, Shuohuan Wang, Chao Pang, Yu Sun, Hao Tian, and Hua Wu. 2022. Ernie-code: Beyond english-centric cross-lingual pretraining for programming languages. *arXiv preprint arXiv:2212.06742*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. 2022. Pangu-coder: Program synthesis with function-level language modeling. *arXiv preprint arXiv:2207.11280*.

Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022. Glm: General language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 320–335.

Michael E Houle and Michael Nett. 2014. Rank-based similarity search: Reducing the dimensional dependence. *IEEE transactions on pattern analysis and machine intelligence*, 37(1):136–150.

Huggingface. 2021. Training codeparrot from scratch. `https://huggingface.co/blog/codeparrot`.

Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547.

langchain ai. 2022. Langchain. `https://www.langchain.com/`.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jimmy Lin. 2022. A proposed conceptual framework for a representational approach to information retrieval. In *ACM SIGIR Forum*, volume 55, pages 1–29. ACM New York, NY, USA.

Jimmy Lin, Ronak Pradeep, Tommaso Teofili, and Jasper Xian. 2023. Vector search with openai embeddings: Lucene is all you need. *arXiv preprint arXiv:2308.14963*.

Sheng-Chieh Lin and Jimmy Lin. 2023. A dense representation framework for lexical and semantic matching. *ACM Transactions on Information Systems*, 41(4):1–29.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Xueguang Ma, Kai Sun, Ronak Pradeep, Minghan Li, and Jimmy Lin. 2022. Another look at dpr: reproduction of training and replication of retrieval. In *European Conference on Information Retrieval*, pages 613–626. Springer.

Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836.

Marius Muja and David G Lowe. 2014. Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence*, 36(11):2227–2240.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Mohammad Norouzi, Ali Punjani, and David J Fleet. 2013. Fast exact search in hamming space with multi-index hashing. *IEEE transactions on pattern analysis and machine intelligence*, 36(6):1107–1119.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Bo Shen, Jiaxin Zhang, Taihong Chen, Daoguang Zan, Bing Geng, An Fu, Muhan Zeng, Ailun Yu, Jichuan Ji, Jingyang Zhao, et al. 2023. Pangu-coder2: Boosting large language models for code with ranking feedback. *arXiv preprint arXiv:2307.14936*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Jing Wang, Jingdong Wang, Gang Zeng, Rui Gan, Shipeng Li, and Baining Guo. 2013. Fast neighborhood graph search using cartesian concatenation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2128–2135.

Kexin Wang, Nils Reimers, and Iryna Gurevych. 2021. Tsdae: Using transformer-based sequential denoising auto-encoder for unsupervised sentence embedding learning. *arXiv preprint arXiv:2104.06979*.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10.

Daoguang Zan, Bei Chen, Yongshun Gong, Junzhi Cao, Fengji Zhang, Bingchao Wu, Bei Guan, Yilong Yin, and Yongji Wang. 2023. Private-library-oriented code generation with large language models. *arXiv preprint arXiv:2307.15370*.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.