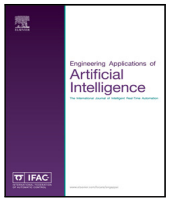
Engineering Applications of Artificial Intelligence 123 (2023) 106304

Contents lists available at ScienceDirect

Engineering Applications of Artificial Intelligence

journal homepage: www.elsevier.com/locate/engappai



DCServCG: A data-centric service code generation using deep learning

Zakieh Alizadehsani [a,b,*], Hadi Ghaemi [c], Amin Shahraki [d,**], Alfonso Gonzalez-Briones [a,e], Juan M. Corchado [a,e]

[a] Air Institute, IoT Digital Innovation Hub, Salamanca, Spain
[b] Faculty of Science, University of Salamanca, Salamanca, Spain
[c] Computer Engineering Department, Ferdowsi University of Mashhad, Iran
[d] Department of Informatics, University of Oslo, Oslo, Norway
[e] BISITE Research Group, University of Salamanca, Salamanca, Spain

ARTICLE INFO

Keywords:
Code auto-completion
Transformers
Language modeling
SOA
Web service
Code mining

ABSTRACT

Modern software development paradigms, including Service-Oriented Architecture (SOA), tend to make use of available services *e.g.,* web service Application Programming Interfaces (APIs) to generate new software. Thus, for the further advancement of SOA, the development of accurate automatic tasks, such as service discovery and composition, is necessary. Most of these automated tasks rely heavily on web service metadata annotation. The lack of machine-readable documentation and structured metadata reduces the accuracy and volume of automatic data annotation, negatively affecting the performance of automated SOA tasks. This study aims to propose automatic code completion for improving web service-based systems by identifying and capturing service usage collected from public repositories that share Open Source Software (OSS). To this end, a Data-Centric Service Code Generation (*DCServCG*) model is proposed to improve old-fashioned, general-purpose code generators that neglect essential service-based code characteristics *e.g.,* sequence overlap and bias issues. *DCServCG* takes advantage of the data-centric concept, i.e., conditional text generation, to overcome the mentioned issues. We have evaluated the approach from the point of view of language modeling metrics. The obtained results indicate that the usage of the data-centric approach reduces perplexity by 1.125. Moreover, the *DCServCG* model uses de-noising and conditional text generation, which is trained on the transformer by distilling the knowledge, DistilGPT2 (82M parameters) trained faster and its perplexity is 0.363 lower than *ServCG* (124M parameters) without de-noising and conditional text generation, which lower perplexity value indicates better model generalization performance.

1. Introduction

The broadness and complexity of software development projects entail high costs in different aspects *e.g.,* monetary costs and human labor. Utilizing available software-reuse techniques in the Software Development Life Cycle (SDLC) can reduce costs and improve productivity. Among these techniques, one of the well-known approaches is Service-oriented Architecture (SOA) (Rochwerger et al., 2009) which focuses on how to assemble software from reusable components. The benefits of SOA become more noticeable when we consider the fact that most software products that run on different devices such as smartphones, and desktops still have common requirements *e.g.,* notifications and file storage. As a result, many companies and organizations, following the SOA concept, provide software-reuse functionalities as online web services. The primary tasks in service-based products can fall into the following steps: (1) Identifying system requirements and required tasks (2) Discovering service registries that usually classify services with the same functionalities for particular tasks, and (3) Developing or composing new service(s). These steps include repetitive tasks which require a lot of effort, however, automation processes can help decrease both time and effort.

In the recent past, service development has demonstrated remarkable success in overcoming the complexities associated with multi-platform software products by employing automated approaches, including rule-based automation or ML-based automation in service discovery (Ma et al., 2018), classification (Zhang et al., 2018), and composing and developing web service (Alwasouf and Kumar, 2019). Most of these studies were conducted on Model-Driven (MD) techniques (Schmidt, 2006), as they rely on high-level abstraction, specify a model once and generate output frequently, which is more understandable for humans. Although MD is highly beneficial in several SOA tasks *e.g.,* service composition, it is adequate for highly repetitive and

* Corresponding author at: Faculty of Science, University of Salamanca, Salamanca, Spain.
** Corresponding author at: Department of Informatics, University of Oslo, Oslo, Norway.
E-mail addresses: zakieh@usal.es (Z. Alizadehsani), am.shahraki@ieee.org (A. Shahraki).

https://doi.org/10.1016/j.engappai.2023.106304
Received 19 October 2022; Received in revised form 15 February 2023; Accepted 6 April 2023
Available online 2 May 2023

Table 1
Review of functional and nonfunctional web service features in the academic and industrial specifications.

Specification	Category	Attribute
Functional	Description	Service name Service architecture *e.g.,* REST, SOAP Service operation list Service technical documentation *e.g.,* WSDL
	Access	Service endpoint URLs Service ports Service accountability Service request limitation
	Interaction	Message exchange pattern *e.g.,* SOAP/HTTP Data exchange *e.g.,* JSON, XML Service request API order *e.g.,* transaction step in payment Service operation list/required params/responses
Non-functional	Performance	Response time Downtime frequency Reliability factors *e.g.,* followers, stars, last update Accountability method *e.g.,* Open-source, Premium
	Security	Encryption Service accountability

predictable tasks, not for a complex system with several scenarios *e.g.,* service code generation (Syriani et al., 2018). Emerging Artificial intelligence (AI) areas (Shahraki and Haugen, 2019), specifically Machine Learning (ML), have opened up new possibilities to solve the challenges of such complex systems (Rafsanjani et al., 2014). The integration of ML and MD tasks has been discussed by a great number of researchers in the literature, where most of the ML-based methods were proposed to improve traditional SOA-based automation techniques. Some studies have taken advantages of the ML-based recommendation systems instead of traditional keyword-based service discovery *e.g.,* (Xiong et al., 2018). Likewise, ML-based classifier models *e.g.,* (Tang et al., 2021) help to reduce service discovery search space by automatically classifying web services.

In spite of the acceptable performance of the models that enhance the efficiency and scalability of SOA tasks *e.g.,* service classification and composition, the accuracy of the mentioned models highly depends on the available annotated metadata of web services *e.g.,* service name, description, URL and port (ProgrammableWeb, 2014). A significant issue in these service-oriented tasks is that they assume metadata for all services is available and easily accessible. This assumption causes a cold-start problem as it occurs when there is not enough data to provide the best results (Wang et al., 2020). Table 1 shows the web service features extracted from the literature (Akkiraju et al., 2005) These features, both functional (how services work) and non-functional (how services should perform), have been used in automation tasks. Although extensive research has been conducted on service specification and SOA-based automation (Cheng et al., 2020), few studies have adequately considered open-source code that uses web service APIs. There are several Open-Source Software (OSS) projects utilizing web service APIs that contain essential information *e.g.,* web service endpoint URLs, port, HTTP models, and the required parameters. A software that provides an analysis of users' behavior based on their activities on social networks can be considered as an example. For this purpose, the software needs to connect to the users' social networks and collect data on their activities. Accordingly, it needs to employ related online services on social networks. After collecting users' data, the software can analyze user behavior using the services provided by Amazon SageMaker endpoints (Fig. 1).

Despite the lack of web service metadata, the source code of projects with web service calls can provide related information *e.g.,* port and URL. Therefore, with the extraction of mentioned part the issue can be transformed into a code auto-complete, including several approaches and methods. In the code auto-complete research area, generative models, in particular, Language Modeling (LM), an active area for generating the next token, demonstrate acceptable results. The earlier code-completion approaches are mostly Natural Language Processing (NLP) keyword-based text representation, that use high-dimensional and sparse source code representation. High-dimensional data representation leads to high resource usage for massive source code datasets. The emergence of word-embedding techniques *e.g.,* static word embedding and contextualized word embedding has improved keyword-based code representation by providing more accurate and low-dimensional text representation. Accelerating open-source solutions in word-embedding techniques *e.g.,* Word2Vec (Church, 2017), BERT (Vaswani et al., 2017) leads researchers to represent more source code data in low-dimension. Consequently, high-scale data provides an adequate situation for using Deep Learning (DL) approaches, which require more data for training. Therefore, researchers utilized different architectures *e.g.,* Recurrent Neural Network (RNN), transformers (Radford et al., 2019) to provide fast and efficient LM. Although these research areas have been investigated heavily, related studies in the auto-complete field are primarily general-purpose code generators that cover greater code variety in target programming languages *e.g.,* Java and Python. These models mainly consider general programming language features in prepossessing and a model-centric training that neglects essential service-based code characteristics such as bias issues (Liu et al., 2021).

This study considers the essential features of web service *i.e.* remote access protocols and endpoint of services during collecting, prepossessing, and training steps. In this way, when developers have less information about calling available web services APIs or parsing the responses, the proposed code completion model can help by suggesting the next tokens according to what it learned from training on source code data. Besides, according to the literature, text generator training is more successful and efficient when the training data is domain-specific (Alsmadi et al., 2022). Therefore, conditional text generation has been utilized to obtain better code auto-completer and overcome sequence overlap problems in web service-based system source code.

In summary, this study has the following contributions:

- The proposal of a model called *DCServCG* for the improvement of service development tasks using auto-completion. *DCServCG* aims to learn web service usage from available OSS source code, not only providing the benefits of code auto-completion, but can also offering the required web service features *e.g.,* endpoint URL and port, while programming.
- The collection of OSS project source code from public repositories. The selected web services are chosen according to the top web service APIs from the primary benchmark dataset in service development tasks called ProgrammableWeb (2014).
- Proposing a method for data de-noising using a binary classifier to remove passive codes and their effects while reducing load

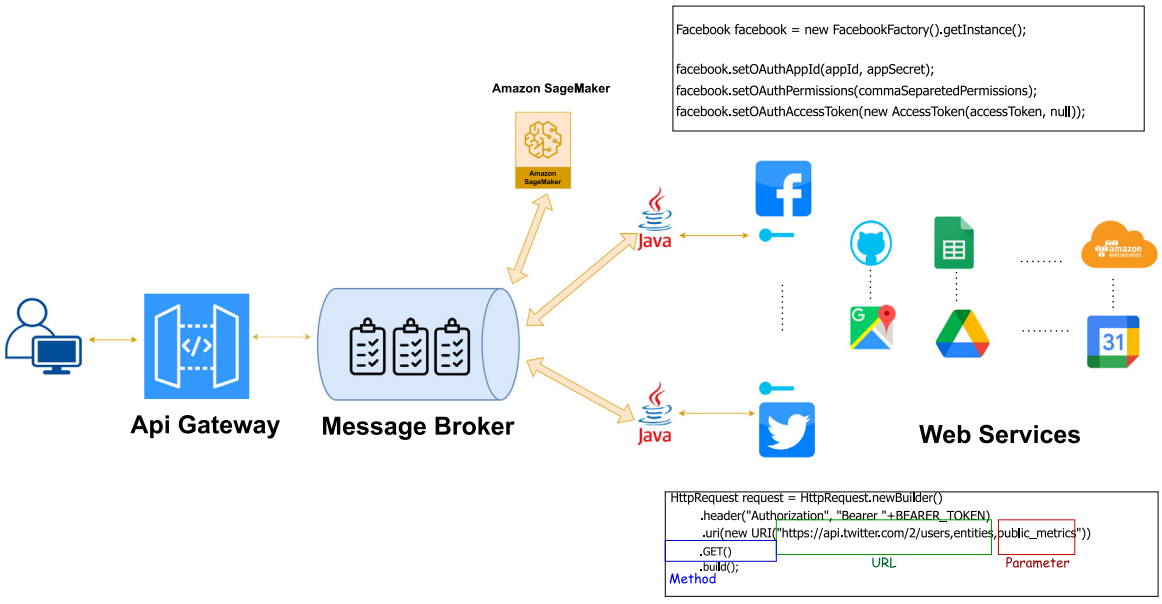


Fig. 1. Java sample code that uses API with calling API end-point.

training time helps to suggest relevant codes including web services usage. This procedure starts with extracting features from the project files with service usage; then, we build a binary-classification model that classifies project files as informative source code or not.

- The proposed data-centric includes a strategy for a conditional text generation approach to overcome web service source code bias and sequence-overlapping issues. The obtained results demonstrate that controlled code generation not only covers rare web service code suggestions but also feeds learning models with well-marginated data to improve results.

The rest of this paper is organized into the following sections. In Section 2, we review the related works. The proposed model is comprehensively defined in Section 3. The experimental results and comparisons are represented in Section 4. Finally, Section 5 describes the conclusion.

2. Related works

This paper proposes an auto-completion model for web services development. Therefore, reviewing the previous works performed in SOA tasks and auto-completion is valuable. Accordingly, this paper split related works into the following sections:

2.1. Service development

In the recent past, service development has demonstrated remarkable success in overcoming the complexities associated with multi-platform software products by using automation in service classification (Zou et al., 2022; Alizadehsani et al., 2022), discovery (Ma et al., 2021a), composing and developing web services (Alwasouf and Kumar, 2019; Wang et al., 2022; Sun et al., 2019; Sangaiah et al., 2020) and web API recommendation (Chen et al., 2021; Wu et al., 2022) etc. Due to web service text features, most recent works in SOA tasks take advantage of text mining, DL-based NLP techniques, and ML. For example, Zou et al. (2022) propose a service classification model called *DeepLTSC*, which aims to address two service classification problems *i.e.* web service multi-dimensional data and unequal distribution of service data. The proposed approach utilizes the Convolutional Neural Network (CNN) model to provide an improved attentive convolutional deep neural network called *LACNN*, which improves feature extraction even in the web services domains including rare web services.

Wang et al. (2022), first, prepare a graph of services, then, they apply the Graph Neural Network (GNN) to extract correlations among services and calculate the probability of utilizing each service to perform the target service-based system task. As the last step, they employ Reinforcement Learning (RL) to improve near-optimal service solutions obtained from the first step. Therefore, if users are unsatisfied with the initial service solution, they can use the proposed method based on Whale optimization algorithm (WOA) (Mirjalili and Lewis, 2016) to improve the initial solution. Tang et al. (2021) utilizes the mechanism of co-attentive representation learning to teach interdependent relationships among service features for service classification. For this, more features are extracted from services using gain theory (Zhang et al., 2018). A Bayesian probabilistic model is presented in Li et al. (2022) for service discovery, which consists of a bi-directional sentence-word model for semantic matching of web service descriptions and queries (words, phrases, sentences, paragraphs).

The mentioned service classification studies demonstrated how service selection could be improved by employing ML-based classification methods. There have also been attempts to define more specific quality parameters to enhance service selection in certain fields. For example, Deng et al. (2016) considers location-sensitive response time as a measure to evaluate the quality of Internet of Things (IoT) (Shahraki et al., 2020) services, and Gharineiat et al. (2021) uses *Spatio-temporal* characteristics of moving IoT services as a quality metric for the composition of services.

In web service development which requires web service usage, code auto-completion can be a valuable pawn to accelerate coding. It is very common to provide models for API recommendation (Chen et al., 2021; Wu et al., 2022). Chen et al. (2021) combine a graph-based statistical model and a tree-based DL model for recommending APIs. In Chen et al. (2021) article, the API structure is specified and reused using graphs, and it focuses more on the development of APIs instead of composing available services. In Wu et al. (2022), authors consider both compatibility and popularity when selecting an API among the candidate items.

In addition to API recommendation, Another API task is generating APIs from a set of source code. In *Deepapi* (Gu et al., 2016), RNN Encoder–Decoder is used to create an API-based on the given query in a natural language. In this study, a binary vector dataset (API

Table 2
Review of deep learning models in SOA and Auto-completion tasks (In header of table CG is Code Generation and ST is SOA task).

Study	Task	ST	CG	Dataset	Application(s)	Metrics	Validation method	Model
CodeGPT-adapted (Lu et al., 2021)	*Code completion Code-to-Code*	–	+	Github Java Corpus/py150	General-purpose development	Accuracy Exact Match (EM)	Hold-out	*LSTM + BPE Transformer (12L) GPT-2*
T5 (Ciniselli et al., 2021a)	Code completion/Code-to-Code	–	+	CodeSearch-Net(Java) Android dataset	General-purpose development Android application development	BLEU score Levenshtein distance Perfect predictions	Hold-out	RoBERTa T5
Codefill (Izadi et al., 2022)	Code completion/Code-to-Code	–	+	ETH150K dataset PY1690K	Structural and name-based information	MRR Accuracy METEOR ROUGE BLEU score	Hold-out	GPT-2
RoBERTa (Ciniselli et al., 2021b)	Code completion/Code-to-Code	–	+	CodeSearch-Net(Java) Android dataset	General-purpose development Android application development	BLEU score Manual analysis Perfect predictions Levenshtein distance	Hold-out	RoBERTa
ServNET (Yang et al., 2020)	Service classification	+	–	ProgrammableWeb	Service selection	Top-1 accuracy Top-5 accuracy	Cross-validation	BERT
Deepapi (Gu et al., 2016)	Service suggestion NL-to-NL	+	–	GitHub and Java documentations	API recommendation	BLEU score	Hold-out	RNN Encoder–decoder
Intellicode (Svyatkovskiy et al., 2020)	Code generation	–	+	GitHub projects	General-purpose development	ROUGE Perplexity	Hold-out	GPT-2
ML+2PN +WOA (Wang et al., 2022)	Service composition	+	–	QWS dataset	Service composition	Solution quality Composition time (s)	Hold-out	GNN
DCServCG(Ours)	Code-to-Code	+	+	ProgramableWeb Service-based OSS Projects Github Java Corpus	Web service development	MRR Perplexity Accuracy Perfect predictions	Hold-out	GPT-2 DistilGPT2

sequence, annotation) was prepared from open-source projects that were extracted from GitHub. In binary vector, API sequence includes the sequence followed to generate the code, and annotation includes descriptions in natural language.

2.2. Code Auto-completion

Several studies have investigated new approaches to code auto-complete for predicting the next token(s), *e.g.,* argument, function, or line, which is used mainly in tools like Integrated Development Environment (IDE). With the emergence of Neural Networks (NNs), current code auto-complete models use different NN architectures, *e.g.,* RNN, and transformers to process code as sequential input data. Some works *e.g.,* Rahman et al. (2020) and Terada and Watanobe (2019) carry out on the applications of RNN and Long short-term memory (LSTM)-based neural networks in code auto-completion. Wang et al. (2019) use LSTM to automatically complete programming codes along with word embedding techniques.

More recent approaches have aimed to overcome the mentioned sequential models *e.g.,* RNN, LSTM limitation. This is because they require sequential computing to perform processes in order, one at a time. In this context, transformers bring parallel computation during the training process; several resources work in parallel, reducing training time. Moreover, transformers transfer knowledge from a well-trained model to enable the training of different models in the same domain problems. In this regard, Ciniselli et al. (2021b) analyze the quality of Bidirectional Encoder Representations from Transformers (BERT), encoder–decoder transformer, for code completion in an empirical study. The authors' findings demonstrate the usefulness of BERT models in predicting the next token in code completion. An example of a pre-trained BERT-based programming language understanding model is CodeBERT (Feng et al., 2020), which is trained on six Natural Language-Programming Language (NL-PL) pairs from the *CodeSearch-Net* dataset (Husain et al., 2019). Moreover, a recent study (Ciniselli et al., 2021a) highlights the usage of transformer models for code completion at different code prediction levels, including single tokens, one or multiple entire statements, and up to a whole code block. The authors' study explores the state-of-the-art pre-trained model, called Text-to-Text Transfer Transformer (T5), which is an encoder–decoder model. They evaluate trained models with metrics, Levenshtein distance (Levenshtein et al., 1966), percentage of perfect predictions, and Bilingual Evaluation Understudy (BLEU) (Papineni et al., 2002) score. Furthermore, the expanded use of transformers has led to various code auto-completion models, some of which are now being used for automatic code generation, such as in Izadi et al. (2022) which proposes the *Codefill*. *Codefill* takes advantage of decoder-only transformers, called Generative Pre-trained Transformer (GPT) models, which combine multitask learning and DL techniques. The authors' proposed approach uses a parallel transformer architecture, which provides accurate code auto-complement. Experiments on Lu et al. (2021) were conducted on 10 tasks along with 14 datasets, including code auto-completion for predicting the line and next token. They investigated several datasets for code development tasks; in code completion, the authors employed two well-known datasets, including GitHub Java Corpus (Allamanis and Sutton, 2013a), and PY150 (Raychev et al., 2016) in Python. They provided *CodeGPT-adapted*, a Transformer-based language model, pre-trained on the mentioned datasets in different Programming Languages (PL). In the auto-complete task, their best results demonstrate 77.73% accuracy in code completion at the token level and 30.60% Exact Match (EM) in line prediction for the Java dataset. Facebook uses two Generative Pre-trained Transformer-2 (GPT-2) and BART (Lewis et al., 2019) models to increase the efficiency and accuracy of automatic code generation in Zhou et al. (2022). Two models were learned using source code files obtained from version control systems. During training, GPT-2 was trained on two different datasets, including auto-completion

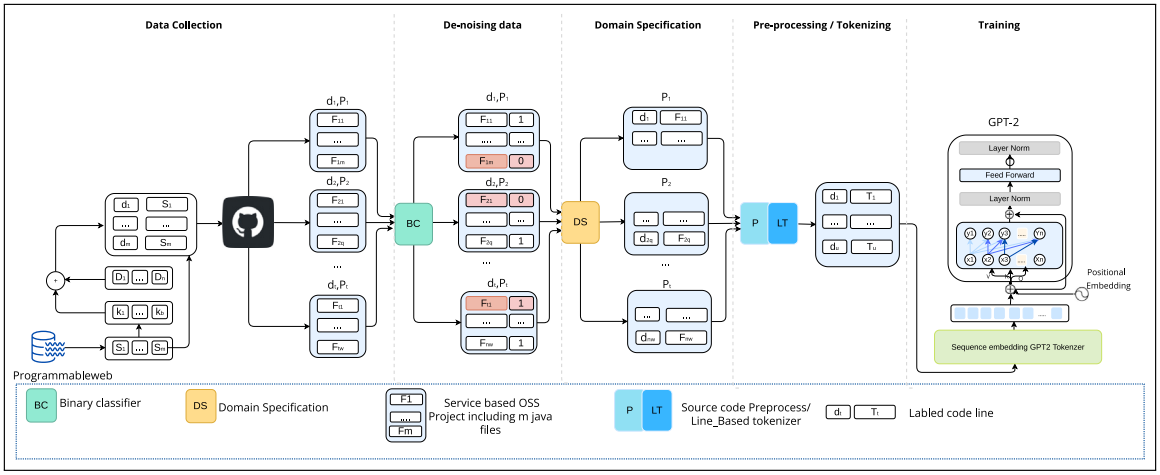


Fig. 2. An overview of DCServCG pipeline.

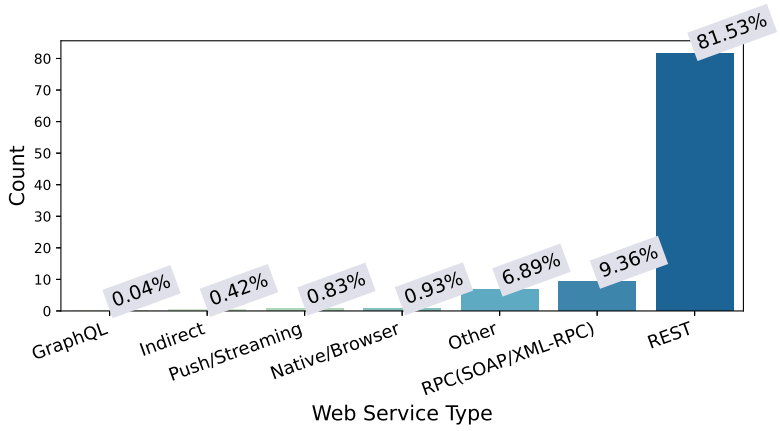


Fig. 3. Distributed services based on the architectural style in ProgrammableWeb demonstrate the service type dominated by the REST approach.

selections which were made during IDE execution and logged code sequences. Table 2 illustrates the review of some deep learning models in SOA and Auto-completion tasks, including utilized model, dataset, performance metric, and validation methods.

Prior SOA approaches apply service metadata as their primary source. In this paper, apart from available services metadata, the source code of service-based OSS projects has been considered. In this way, DL models can be employed to extract service usage patterns automatically. The extracted patterns help developers to use web services in the form of auto-completion. Moreover, this work empowers the auto-completion model with data-centric approaches to obtain more accurate results.

3. Model structure

This section describes the proposed model, *DCServCG*, which employs both data-centric (Belinkov et al., 2019) and transfer learning (Pan and Yang, 2009) concepts for improved service-based development. The presented method architecture is divided into three parts: (1) source code collection, which utilizes the top web service APIs from ProgrammableWeb (ProgrammableWeb, 2014); (2) iterative application of data-centric techniques *e.g.,* de-noising, and domain specification, to optimize data; and (3) the training of a service code auto-completion model employing a transformer learning technique by fine-tuning the pre-trained GPT-2 model. Fig. 2 demonstrates the architecture of the proposed method including data collection/optimization and training of the DL-based model for service development tasks. The details of the model structure are explained in the rest of this section.

3.1. Web service source code collection

For *DCServCG* training, preparing a set of source code as datasets that use web API is required. The usual approach to source code dataset collection, called code mining (Allamanis et al., 2018), is the use of popular repositories such as GitHub and Stack Overflow. However, the datasets are generally prepared for SOA tasks as they are mostly based on web service metadata. For example, ProgrammableWeb, a well-known dataset for SOA tasks, includes web service metadata. To this end, the data collection process for extracting source code is divided into two steps. First, the top web APIs from ProgrammableWeb are extracted, and then the OSS projects that use these web APIs are gathered from open-source repositories. $S = \{s_1, \ldots, s_n\}\ 0 < n < N$ stands for the services APIs extracted from ProgrammableWeb, where N represents the number of services, s_j represents jth service that is defined as $S_j = (T_j, F_j)$. $T_j = \{t_{j1}, \ldots, t_{jp}\}$ represent the set of tags assigned to the service jth. $F_j = \{name_j, description_j, followers_j, developers_j\}$, as additional information about the service, including metadata and nonfunctional features of the jth service.

In the same way, set $D = \{d_1, \ldots, d_m\}\ 0 < m < M$ indicates the domain extracted from the ProgrammableWeb. In this context, $d_j = (name_j, project_j)$ represents jth domain, $name_j$ is the domain's name, and $project_j$ stands for the number of projects extracted from open-source repositories based on the jth domain.

$|S|$ and $|D|$ demonstrate selected services and domains respectively; Moreover, f_{ith} and d_{ith} indicate non-functional quality features of services *i.e.* followers and numbers of developers that used the service respectively. In addition, D' stands for the top domains with the most

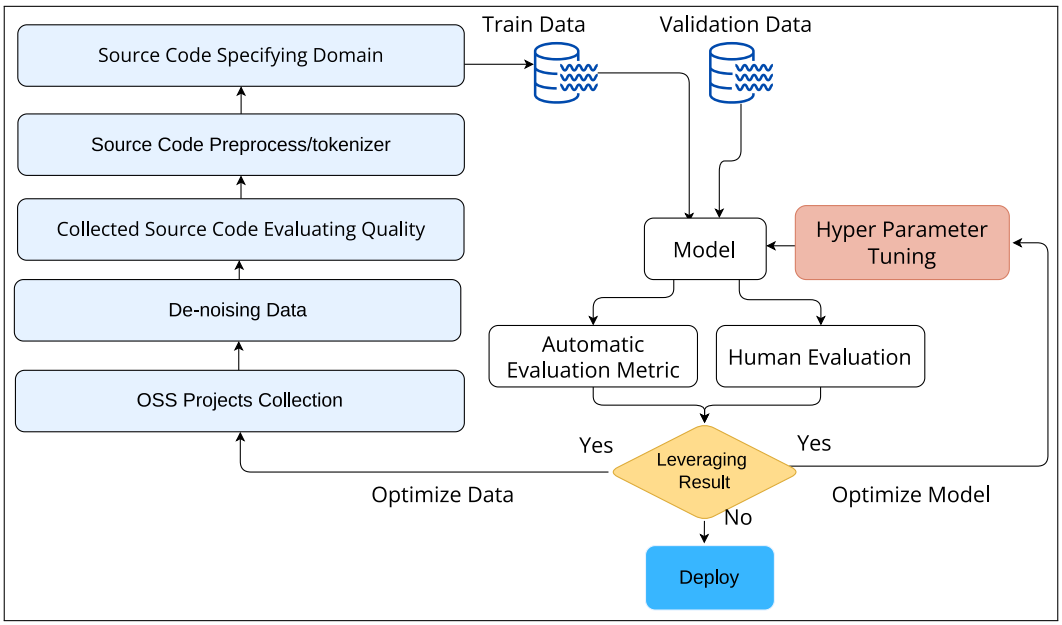


Fig. 4. DCServCG aims to take advantage of both model-centric and data-centric methods. There are two loops in the proposed step; the first loop includes interconnected chained steps for the optimization of training data systematically, and the second loop is related to the tuning model.

Table 3
Some illustrative APIs and associated endpoint URLs examples.

Service API	Web service endpoints
YouTube	https://www.googleapis.com/youtube/ https://www.googleapis.com/auth/youtube https://www.youtube.com/user/haguden/videos https://www.googleapis.com/<token> ?playlistId=<id>&key=<token>
Amazon product advertising Instagram graph	https://api.rainforestapi.com https://graph.facebook.com/
Facebook	https://graph.facebook.com/ https://graph.facebook.com/<id>/picture?width=<num>
Google Map	https://maps.googleapis.com/map https://maps.googleapis.com/maps/api/staticmap https://maps.googleapis.com/maps/api/js?key=<token>

services, and T' defines the top tags that are used in services. D select n top values between T', D' and S, defining services with top scores based on $|S'|$.

S' and $|S'|$ are defined as:

$$|S'| = \{(f_1 + d_1), (f_2 + d_2), \ldots, (f_n + d_n)\}, s_o \in S', \exists T_{op} \in T_o, T_{op} \in D^* \quad (1)$$

Consequently, the projects that include the services in S^* are extracted from GitHub. Given that these projects are defined as $P = \{p_1, \ldots, p_q\}$, the $p_w \in P$ represents wth project that is defined as $p_w = \{Files_w, Domain_w, Star_w, Fork_w\}$. Where $Files_w$ is a set of project files. $Domain_w$ represents the project domain, $Fork_w$ and $Star_w$ are quality characteristics of the project that describe the project stars, and the number of forks is taken from the project. The stated process is indicated in Fig. 2 as the data collection step.

It should be noted that Representational state transfer (REST) is currently the most used architecture for developing web services; Therefore, the collected OSS projects use top APIs with REST architecture. Fig. 3 shows the different types and related distribution percentages in the ProgrammableWeb dataset, in which more services have implemented REST. Table 3 provides examples of associated web service endpoint URL. Most URLs are related to top APIs that provide data extraction/online streaming *e.g.,* YouTube, Twitter or functionality sharing *e.g.,* Google map direction functionality. Accordingly, the current paper collected service-based Java projects from GitHub, which utilized REST APIs. The collected projects and code lines are paired with associated web service domains. The statistics collected from the projects are indicated in Table 4. For each of these service-based projects, the metadata of the project *e.g.,* name, git repository, and quality metrics, are listed.

3.2. Data-centric Pipeline

ML models are typically created through two main phases, (1) the preparation of data and (2) the construction of learning models. Research on ML has largely focused on the models, which has involved selecting and improving them with a wide range of algorithms and techniques *e.g.,* Shallow Learning, Deep Learning, Active Learning, Transfer Learning, Ensemble Learning (EL). Currently, with the increasing number of DL transfer learning models, which are highly data-demanded, the importance of data has gained attention in a topic called data-centric (Bashath et al., 2022). In the data-centric approach, data are systematically modified or enhanced to improve the performance of the system. As opposed to the model-centric approach, this time the model is fixed, and only the data needs to be improved. Smart data labeling and de-nosing are instances of systematic data-centric techniques employed in this work. Fig. 4 shows the defined data collection and de-nosing performed iteratively until the dataset is balanced for most web service domains. For this purpose, after each data collection operation, a de-nosing operation is performed, which is associated with

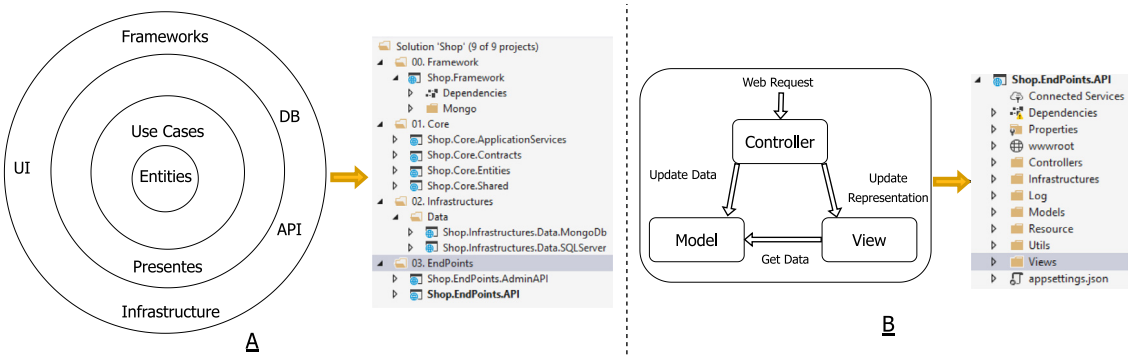


Fig. 5. (A) Sample of clean architectures and structure of the source code files, (B) Sample of MVC architectures and structure of the source code files.

Table 4
Statistics on collected web service codes and GitHub Java Corpus subset (Allamanis and Sutton, 2013b).

Items	OSS Web service codes train/test	GitHub Java Corpus
Files	216,399/58,158	141 606
Lines of Codes(LoC)	20,028,866/6,445,062	19,354,856
Tokens	376,983,686/123,687,908	375,542,147

the removal of noisy files. After checking the balanced status of the domains, the described process is executed again if needed.

It can be seen in Fig. 4 that the defined data pre-processing and de-noising in a loop may also break out of the loop depending on the model's performance metrics. In this study, data-centric actions involve a series of interconnected data optimization activities that mainly involve minimal human interaction. The principal steps of proposed data-centric pipelines are (1) data collection (2) selecting/filtering/de-noising data, (3) preprocessing and tokenizing, and (4) evaluating data quality. The details of the data-centric pipelines are explained in the rest of this section.

3.2.1. Selecting/filtering/de-noising data

Considering the new data-centric concept, using informative data is more effective than using large quantities of data. In this way, detecting reusable source-code based on design pattern (Gamma et al., 1995) concepts is valuable. Depending on the design patterns and architectures used for implementing programming projects, the code of these projects is composed of different parts that separate project source code into code area *e.g.,* source code for working with data, source code areas for providing internal services, and source code areas for using passive and active classes. Using external services has a significant impact on software development since most of these parts cannot be developed internally in the software or are expensive to develop. Thus, we consider design patterns and architectures as informative factors. Over time, comprehensive literature has evolved on design patterns that have categorized software architectures into three or four layers based on software layering. An example of this would be the Business Logic Layer (BUS) of a three-layer architecture, where developers can call pre-prepared web services (Martin et al., 2018). For instance, two widely used architectures in software development, Clean (A) and MVC (B), are shown in Fig. 5 along with the structure of the source code files. Depending on the architecture used for software development, system components are located differently that have different structures. Thus, it is difficult to detect and identify active source codes. In this case, it becomes more difficult when we realize that many software projects have used their own, unknown architectures. Therefore, it is valuable to provide a model for the automatic identification of active source codes that focus on code data.

In the current study, noisy data refers to source codes that do not use services or well-known libraries. For example, the following codes demonstrate source code from a class in Kubernetes.[1] Although projects like Kubernetes has high-rate star on GitHub, they contain several codes which do not use third-party libraries or web services. These codes mainly include defining data structures, interfaces, etc. Moreover, these classes have been used repeatedly and can cause problems in the learning process of the code auto-completion model. In this study, an automated de-noising strategy is used to feed the learning model with more relevant data.

Java OSS Project Sample Code

```
package io.kubernetes.client.openapi.models;
import io.kubernetes.client.fluent.*;
  public AuthenticationV1TokenRequestBuilder
  (Boolean validationEnabled) {
    this(new AuthenticationV1TokenRequest(),
    validationEnabled);
  }
```

To address the mentioned issue, an automated classifier model has been developed on the basis of the contents of the Java file, to extract informative parts from the software project code. The first step of the model is to extract some features from each Java code file in the service-based projects.

Regarding the mentioned formulation, the classifier's input is a file of the collected OSS projects. These project files $P = \{(f_{11}, \ldots, f_{1x}), \ldots, (f_{n1}, \ldots, f_{nz})\}$ are given to the binary classifier. For P, the output from the binary classifier is a set of labels associated with the projects file $L = \{(l_{11}, \ldots, l_{1x}), \ldots, (l_{n1}, \ldots, l_{nz})\}$. The binary classifier output determines whether a file label is (0) or (1); then, $P.L$ in Eq. (2) feeds GPT-2 model as an input for training.

$$P.L = \{P_1(fl_{check}(f_{11}, l_{11}), \ldots, fl_{check}(f_{1x}, l_{1x})), \ldots, P_n(fl_{check}(f_{n1}, l_{n1}), \ldots, fl_{check}(f_{nz}, l_{nz}))\} \quad (2)$$

$$fl_{check}(f, l) = \begin{cases} If \quad l = 0 \quad return \quad null \\ If \quad l = 1 \quad return \quad f \end{cases} \quad (3)$$

Feature Selection: The extracted features are based on the source code of Java files, which are listed below:

- Number of classes in file: It is common for Data transfer object (DTO) classes and data structures to be contained within one file rather than active classes that use online services.
- Several packages or classes are used in the Java file: Two types of classes defined in software projects are Active and Passive. Active classes are packages/classes that have operational behavior and usually make changes to the data, while passive classes are classes that are defined to declare a piece of software. Using the

[1] Kubernetes https://github.com/kubernetes/kubernetes.

definition provided, this study tries to identify Active classes and the number of Active packages/classes used in the file which is directly related to its probability of being Active.
- The number of classes using internal classes: Passive classes are usually the inner classes, which are typically declared private and used in the target project scope.
- The number of classes used from external project classes: It is desirable to have classes that are not only Active but also use external services.

Binary Classifier After extracting the features, they are given to the binary classifier. In binary classifier, 0 indicates the lack of usage of external services, and 1 means that the source code has an acceptable degree of software usage and uses external services and libraries. For this purpose, several ML classifiers *e.g.,* Support vector machine (SVM), naive Bayes, and gradient boosted models, have been used. Finally, the project files were classified and the files labeled "1" were given to GPT-2 for learning. Section 4.2 discusses the impact of this strategy on the final results.

3.2.2. Preprocessing and tokenizing

Following code collection, prepossessing and tokenizing code is essential to making code suitable for the training process, which can increase the performance of learning models. Some earlier approaches have used NLP techniques (*e.g.,* segmentation, tokenization, removing stop-words, stemming) (Allamanis et al., 2018). Nevertheless, source code is a kind of structured text data, and according to the literature, NLP techniques do not achieve high performance. The main reason is that these techniques suffer from ignoring the concept of code structure. Therefore, the community has considered prepossessing steps based on code structure, such as filtering and de-duplicating source code, removing markup languages and metadata, removing comments and common sections, and *CamelCase splitting*. Moreover, particular prepossessing is needed to normalize developer-defined strings and numbers in source code which has been done in previous works (Lu et al., 2021; Izadi et al., 2022). Zhou et al. (2022) have defined particular criteria according to their research goals, such as removing functions with fewer than three lines and eliminating methods with more duplication using the de-duplication algorithm in CodeSearchNet. Our study has done preprocessing in two stages; first we clean projects from unnecessary files with different types *e.g.,* JSON, YAML, or patch files (files containing output from diff commands). Afterward, we clean the java file content to have only java codes. Javalang (2019) is utilized for this step, providing a lexer and a parser for the Java language. Javalang extracts java source codes from files that can automatically remove comments. Moreover, this parser facilitates normalizing developer-defined strings and numbers in the source codes.

A major drawback of general-purpose code preprocessing is that URLs have been assumed as normal strings and replaced with string placeholders. In service development, URLs mainly include service endpoints in service development, and simply removing them would cause the loss of a lot of valuable data. In contrast to the mentioned studies, current work keeps web service API URLs and normalizes literals URLs. Each service provider generally offers different services on specified domains and ports. These services have different routes and parameters that identify each section containing special meaning. To keep web services data, a set of data from domains of services are extracted. This collection is used to identify the web service endpoint in the source code; Moreover, the customized parameters of web APIs are preprocessed before feeding the LM model. For example, *"<STR_URL>", and "<STR_TOKEN>",* respectively replaced the values of none-service URLs and API tokens (access tokens which allow developers to authenticate to APIs).

Following code collection and preprocessing steps, the tokenization of OSS projects is a required step which split projects into smaller units. It is common to tokenize source code projects into smaller units *e.g.,* files, methods, APIs and code lines during data mining. In this study, the tokenization process can be considered as $T = F(P\_i)$, in which T is a tokenizer that receives a project P_i as an input and performs tokenization based on the files or lines of the project using Java lexer and parser.

3.3. Conditional source code generation

In recent years, well-known service providers have published various web services that help developers design and develop software. These web services can be in diverse fields with different applications. For example, Google Cloud provides a wide range of APIs for various online services in domains such as Google Maps, Google Calendar, or YouTube. The similarity of the endpoint URL of these web services can confuse the auto-completion models and affect the model's accuracy. On the other hand, imbalance in the usage of web services usage can have a negative impact on the training process in different service domains. For example, Google Cloud APIs provides different domains of services and Google Maps APIs are more utilized in OSS projects than Google Calendar APIs. Consequently, in collected software projects, this leads to imbalance the dataset, which can discard valuable information about some web services with minor source code.

In recent years, there have been some works for controllable text generation (Alsmadi et al., 2022) and topic-aware text generation (Ma et al., 2021b), which can deal with imbalanced data issues. In this context, *DCServCG* uses the metadata of web services to determine the domain of each web service, providing controllable code auto-completion. Given that the smallest unit of input are lines of code lines, each file is defined according to its lines as $F_p = (ln_{p1}, ln_{p2}, \dots, ln_{pg})$ and its domains as d_p, so the output of each file will be $\bigcup_{i=1}^{g}(d_p + t(F_p[ln_{pi}]))$, and g is the number of lines in the F file. If m is the number of data files, $|F_p|$ is the number of lines and d_p is the domain(s) of the F_p file, the tokens created for the dataset will be equal to $\bigcup_{j=1}^{m}\bigcup_{k=1}^{|F_j|}(d_p + t(F_j[ln_k]))$.

3.4. Training the DCServCG auto-completion model

Previous studies *e.g.,* Alsmadi et al. (2022), Lu et al. (2021) and Izadi et al. (2022) have shown that transformer models such as GPT-2 can be appropriate for auto-completion tasks. The decoder transformer predicts the next token on the basis of self-attention and the available token input sequence. The REST of this section discusses the preprocessing and training phases in depth. Besides, to clarify the research problem and the expected results, the research algorithm is demonstrated in Algorithm 1.

3.4.1. Training the model

Regression (Seltman, 2012) is one of the classic algorithms, which usually uses a linear model to establish the relationship between dependent and independent variables. Eq. (4) represents a simple linear regression, where Y is the dependent variable, X stands for the independent variable, which indicates possible values of Y when X is restricted to some single values. In Eq. (4) β_0 stands for a point where the graph crosses the y-axis, and the slope parameter is β_1.

$$Y = \beta_0 + \beta_1 X \tag{4}$$

Eq. (4) demonstrates a simple regression, which only considers a single explanatory variable. *Multiple regression* is a type of regression for analyzing the relationship between a single dependent variable and several independent variables using a linear combination of predictors. Another regression model type is Autoregressive (AR) models, which are employed in language modeling tasks and predict future values using only past values. By definition, in Eq. (5), the AR model predict Y_i on the basis of previous values (y_{t-1}, y_{t-1}, $\dots$, y_{t-p}), where p is a variable that indicates the order of the AR. To this end, text generation isn based on AR *e.g.,* decoder-only transformers can only see the text's left side.

$$Y_i = \beta_0 + \sum_{i=1}^{p} \beta_i y_{t-i} + \epsilon_i \tag{5}$$

Algorithm 1 Data-centric and Training Process

Require: $n, j, z \geq 0$
$Input : S = \{s_1, .., s_n\}$ Services APIs extracted from ProgrammableWeb
$S_j = (T_j, F_j)$
T_j, Assigned tags for the jth service
F_j Assigned additional metadata for the jth service
$Output : P = \{p_1, ..., p_q\}$, OSS service-based Projects
$p_w = \{Files_w, Domain_w, Star_w, Fork_w\}$
Function `ColectData(`S`)`:
  for $item_s$ *in* S **do**
    if $item_s$ *in* $|S'|$ **then**
      $p \leftarrow$ collectOSS projects $(item_s)$
      P.append(p)
    end
  end
return P
$Input : P = \{p_1, ..., p_q\}$
P_{SC} demonstrate projects source code
$Output : Dataset : \{LoC_i, .., LoC_t\}$, $Dataset : \{file_i, .., file_t\}$
Function `DCProcess(`SC,$tokenize_{type}$`)`:
  $P \leftarrow Deduplicate - projects(P)$
  while *Debugging* **do**
    $P_{SC}.ExtractTargetFiles()$
    $P_{SC}.DataQualityAssessment()$
    if *De-noising* **then**
      $P_{SC}.Denoising()$
    end
    $P_{SC}.CodeLexerParser()$
    $P_{SC}.NormaliseDefinedValue()$
    $P_{SC}.NormaliseServiceEndPoint()$
  end
  $Dataset \leftarrow P_{SC}.Tokenize(tokenize_{type})$
  $P_{SC}.DomainSpecification()$
  $TrainSet, TestSet \leftarrow P_{SC}.Split()$
  return *Dataset*
$Input : Train_{Set}, Test_{Set} : \{LoC_i, .., LoC_t\}/\{file_i, .., file_t\}$,
$SourceModel\ type : SM_{type}$
$SourceModel\ Hyperparam : HP_m$
$Output : M_{generator}, Trained Auto - complete model$
Function `Training(`$Train_{Set}$,$Test_{Set}$,SM_{type},HP_m`)`:
  $model_{type} \leftarrow GPT - 2\ \ model\ M_{generator} \leftarrow Finetune(model_{type}, Train_{Set}, Test_{Set}, HP_m)$
  return $M_{generator}$

Previous works in both industry (TabNine, 2019) and academia (Lu et al., 2021) show that decoder-only transformers which use the auto-regressive idea, such as GPT-2 may be appropriate for auto-completion tasks. In the *DCServCG*, decoder transformers are added to the sequence of inputs once each token is predicted; therefore, the new predicted sequence is the input to the model in the next step. If the samples stand as $x_1, x_2, \ldots, x_T$ formed of variable symbol-sequence lengths, the sentence prior probability $P(x_{1:T})$ is often calculated using the conditional probabilities in Eq. (6):

$$P(x_{1:T}) = P(x_1) \prod_{t=2}^{T} P(x_t | x_{:t-1}) \tag{6}$$

Additionally, transformers models contain multi-head attention modules with self-attention. This multi-head attention includes parallel heads that take advantage of the attention mechanism called *"Attention is all you need"* (Vaswani et al., 2017), which has been introduced by Google.

Most of the recent language models that use the attention mechanism show acceptable performance in general-purpose programming; Regardless, the impressive results achieved by current transfer models need to consider some factors such as training time, model size, number of parameters, and token(s) prediction time. Selecting a large pre-trained model can be demanding in terms of memory and computational resources. Among the presented transformer models, the GPT-2 as a decoder-only transformer, has a significant performance in both industry and academic communities (Izadi et al., 2022). GPT models have also been proposed in several versions of pre-defined variants: distilled, base and large, which are different in size, model parameters, and training time. Distilled-GPT2 (DistilGPT2), the distilled version of GPT-2, has 82 million parameters, GPT-2 medium has 124 million parameters, and Generative Pre-trained Transformer 3 (GPT-3), which is the newer version, has 175 billion parameters. Regarding the encoder–decoder transformer, T5 also had a significant performance in the encoder–decoder models (Ciniselli et al., 2021a); however, the t5-small with 60 million parameters need more computational resources than DistilGPT2 with 82M parameters. Despite T5 or GPT-3 providing higher accuracy, these models require more system resources *i.e.* RAM and CPU and time. Taking into account the accuracy of the model and the trade-off between run-time, this work implementation structure used GPT-2 base and DistilGPT2.

As mentioned before, a service provider can release various services that cause similarity in service usage in the source code. Consequently, some parts of the training data may have gray areas that lead to overlap in the data, negatively affecting model learning. In this context, controlled code generation can feed learning models with well-marginated data to achieve better results. With respect to the mentioned definitions, assume the Web service domains set $D = \{d_1, d_2, d_3, \ldots, d_m\}$; and train the code lines string as $C = \{c_1, c_2, c_3, \ldots, c_n\}$. The training dataset for conditional code generation is a one to one concatenation of two sets, D and C, which were obtained during data collection. Given a set of service domains defined as $D = \{d_1, d_2, d_3, \ldots, d_m\}$, and m > 0 is the number of service domains. According to language modeling Eq. (6), conditional code generation can be defined as:

$$P(Y|C, D) = \prod_{t=1}^{T} P(y_t | C, D, y < t) \tag{7}$$

In Eq. (7), C stands for $c_{input} = \{[d_{ith}], c_{i0}, c_{i1}, c_{i2}, c_{i3}, \ldots, c_{iT}\}$. Here, ci_t means the ith code token that the developer has written in the IDE editor. In this study, the auto-complete code problem employs language modeling models. The objective is to guess and calculate $c_{it} + 1$.

4. Experimental results

In this section, the performance of the proposed models is demonstrated. As stated before, the proposed architecture includes components that aim to improve results. To evaluate the impacts of those different components in the proposed code auto-completion performance, we separate the architecture and explored the effects of each component. To this end, this study uses the Ablation study, which aims to examine the performance of each component of the AI system by removing the target component to evaluate the effectiveness of the components (Izadi et al., 2022; Xu et al., 2021).

In this process, several models have been trained, Table 5 shows the components used by the models. As shown in the table, in the current study *ServCG* is the baseline model. Moreover, two types of code tokenization are investigated on source code projects to analyze the results including Line-based and file-based types.

Subsequently, to investigate the quality of the collected web service, we compare it with the GitHub Java Corpus (Allamanis and Sutton, 2013b), which is a benchmark dataset for code mining tasks in Java language. Therefore, this benchmark dataset lets us compare the collected source code that used web services with general-purpose code. Table 4 demonstrates the information about the detailed statistic for both datasets.

4.1. Training process

This section describes the training environment and metrics while training which can give valuable information about learning performance and resource usage over time.

Exploring the training data can help to identify the data structure, facilitating the building of models with the correct parameters. Fig. 6 illustrates the density plot for collected code length and token distribution. In the plot, (a) and (b) show the size density of tokens according to line and (c) and (d) according to the file. Data distribution density

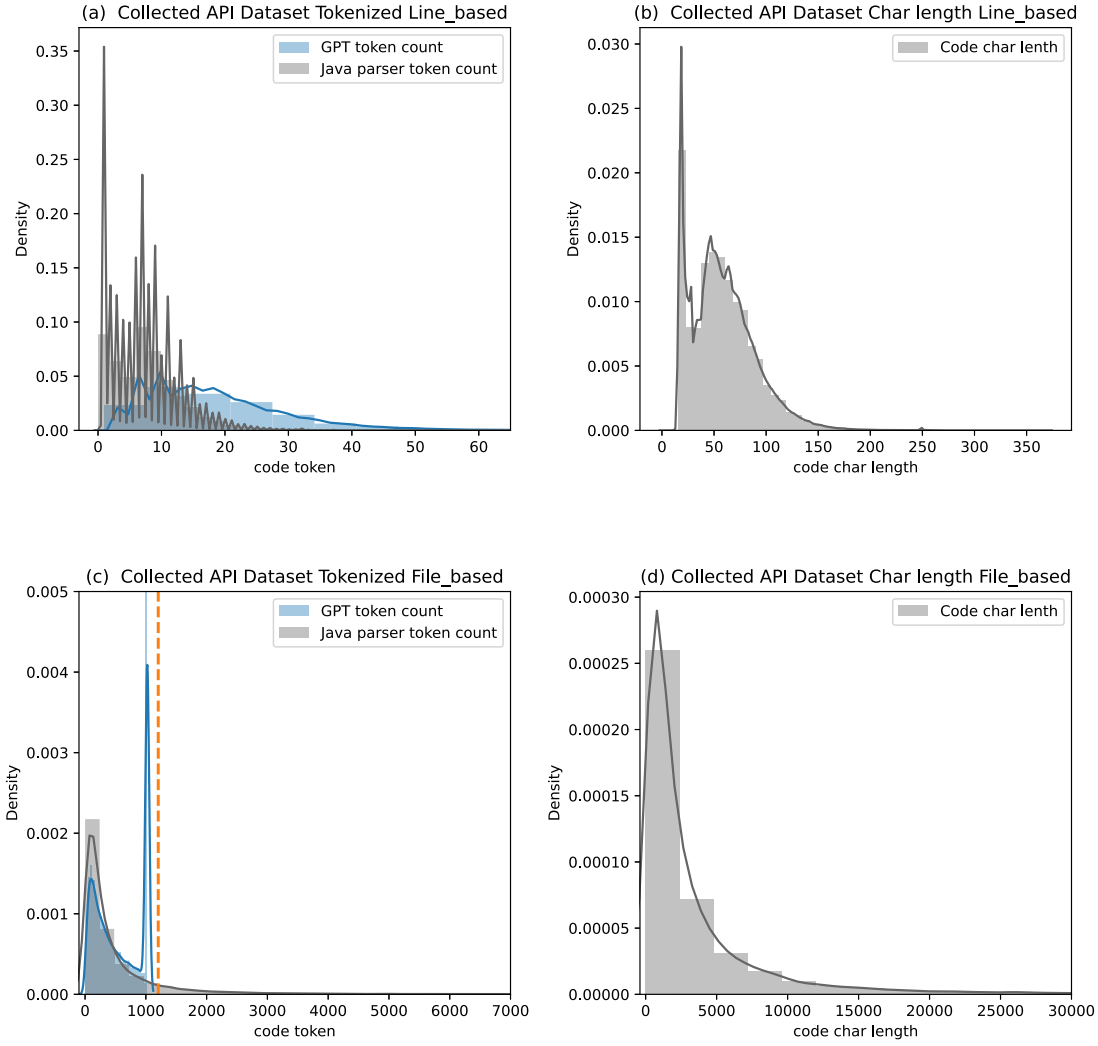


Fig. 6. Density plot to represent the distribution of the length of the collected codes and the tokens.

Table 5
Models structures.

Model	Transformer	Tokenization	Components
ServCG	GPT-2	Line/file	Code collection 3.1
ServgCG_den	GPT-2	Line/file	Code collection 3.1 De-noising 3.2.1
DCServCG	GPT-2 DistilGPT2	Line/file	Code collection 3.1 De-noising 3.2.1 Conditional generation 3.3
CG_g	GPT-2	Line	Dataset (Allamanis and Sutton, 2013b)
CG_den_g	GPT-2	Line	Dataset (Allamanis and Sutton, 2013b) De-noising 3.2.1

is specified according to both GPT tokenizer and Java parser (Fig. 6(a) and (c)) and the number of characters (Fig. 6(b) and (d)). The GPT tokenizer density distribution state is marked with blue, and the Java parser with gray color. The density plot in section (a) indicates that more data plots are found in the range of 0–60, while their largest density value is different in both plots. The largest density in Java parser is 0.35, and in GPT tokenizer it is 0.05. Also, GPT tokenizer density has a more normal distribution.

GPT-2 tokenizer provides a full 1024 tokens context window. This value is enough to tokenize the prepared dataset according to the lines of code. Therefore, based on code lines length distribution in Fig. 6, the present study has selected $max\_length = 100$ to tokenize a line-based dataset. The GPT-2 tokenizer maximum value does not support the maximum coverage for the dataset prepared on the basis of Java files. GPT-3 has decreased this token limitation to 4000 as of June 2021, however, regarding resource limitation, this study has developed a pipeline for GPT-2. Besides, as demonstrated in the 6, section (c), the

Table 6
The characteristics of the system.

Items	Characteristics
Computing platform	Google Colab Pro+
GPU	P100, T4
RAM	50 GB RAM
Python libraries	HuggingFace, Scikit-learn

distribution has skewed right, and fewer data plots are found with code tokens higher than 1024.

Utilizing DL-based transformers for code auto-completion required tuning DL hyper-parameters such as epochs, batch size, and learning rate. In addition, to find the optimal value for these parameters, not only factors such as model validation results are important, but according to the size of these models (124M, 85M), resource usage *e.g.,* memory should also be considered.

Accordingly, to control the learning process, the hyper-parameters are tuned at learning rate = 1e−4, with Adam optimizer (Lydia and Francis, 2019), which is a stable optimizer with faster computation time and fewer parameters demanding for tuning. In the training process, overfitting/underfitting prevention is considered at the data level and by selecting suitable model hyperparameters. In the data level, apart from increasing training data to cover selected domain service usage variety, we try to consider the same distribution of data in the training and testing data. In the case of model hyperparameters, the $hidden\_dropout\_prob = 0.1$ in GPT models is defined, which is responsible for the dropout probability for all fully connected layers. In hyperparameter selection, moderated batch size and epochs are considered based on the recommended value for the large datasets by OPEN API (OPENAI, 2018) to avoid overfitting. Respecting the epoch, the recommended value is 4 epochs for general generation tasks and 2 epochs for conditional generation. Therefore, in this study, we have selected $epochs = 2$ that avoid over-fitting or forgetting sequence data problems in our conditional generation approach. Another parameter is batch_size, which determines the batch that is processed in parallel, and prevents DL models from only focusing on global optima in a dataset. The batches' size value selection is directly proportional to dataset volume and variety. In general, the small size of batches helps to learn better in small data; However, the choice of a small batch size for larger datasets like our dataset (including local code) not only increases resource usage and learning time but also can add noise to the learning process. To this end, batch_size is tuned on Learning $batch\_size = 32$. The characteristics of the system and the resources for training and evaluation process are summarized in Table 6.

4.2. Investigating the impact of de-noising strategy

DL models tend to over-fit models with complexity or long-time training. In this way, the model attempts to fit the training data entirely and memorizes the data patterns, including repeated noises. Therefore, if there are some repeated passive source code in big projects (code 3.2.1), then the model learns that this code introduces noise and is not reusable.

As stated in Section 3.2.1, among the extracted source code, the files that contain active classes are suitable options, and the removal of passive type source code is the purpose of the section related to de-noising. For this purpose, a dataset has been selected among the extracted source code, and the type of source code (active, passive) has been specified manually. This collection contains 16 015 source code files, 8933 of which are labeled as active and the REST are labeled as passive. Several classifiers have been examined by this dataset. Fig. 7 shows the performance of different classifiers. According to the binary classification results, the Linear discriminant analysis (LDA) model performs better with (78.54%) accuracy than the other models. Accordingly, the classifier LDA has been used to identify and remove

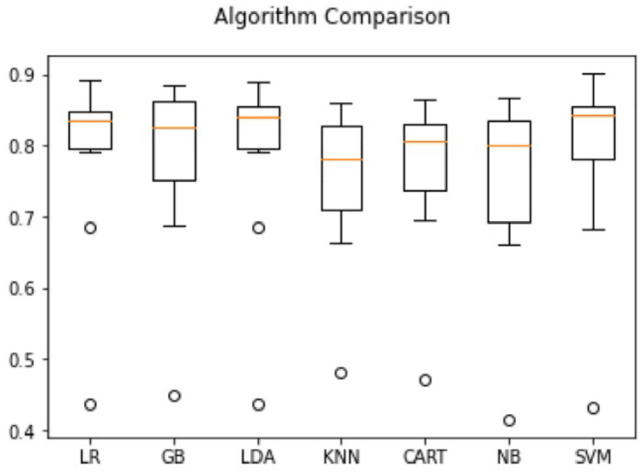


Fig. 7. Result of binary classifier for de-noising.

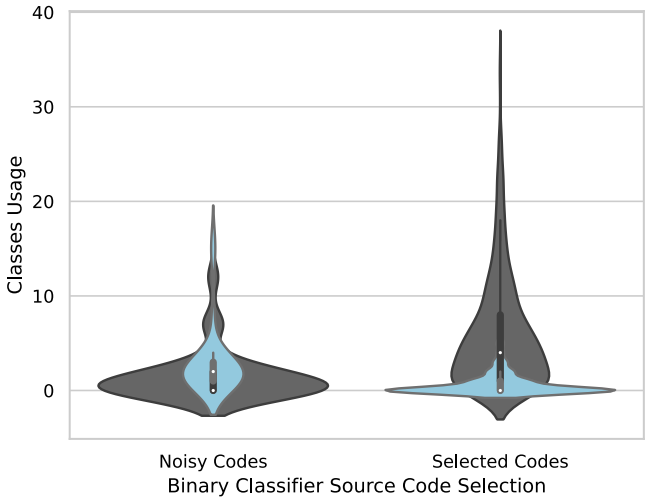


Fig. 8. Passive and Active classes distribution in de-noising binary classifier results. Passive classes are marked in blue and the Active classes marked in gray.

noise (passive classes). Fig. 8 demonstrates the effectiveness of Passive and Active class distribution in de-noising binary classifier results. This figure indicates that the selected code includes more active classes and fewer passive classes.

The proposed de-noising strategy removes 26.96% of noisy data, increases the performance of the training process by reducing 31.02% of training time and resource usage (Table 7), with only 0.12 rise in Perplexity (Table 8). This minor rise of perplexity is mainly related to eliminating passive files, including general programming operations, such as defining variables and using loops. In addition, there are sections with general operations in active files. As a result, the elimination of passive source code files can affect available sections and minor rises in perplexity of this part of the source codes.

To provide more analysis of the performance of the proposed de-noising strategy, we offer a random training file selection for training data. This file selection considers fair random file selection to keep model generalization by random file selection from each domain. The training time was similar to the denoising strategy, but obtained perplexity for random file selection is perplexity = 3.0243, which increased remarkably compared to denoising perplexity = 2.622 with the same volume of data. This lower perplexity in denoising indicates that de-noising provides more useful codes, which improves auto-completion generalization performance. However, the main benefit of code de-noising is that the auto-complete model can feed with codes, including

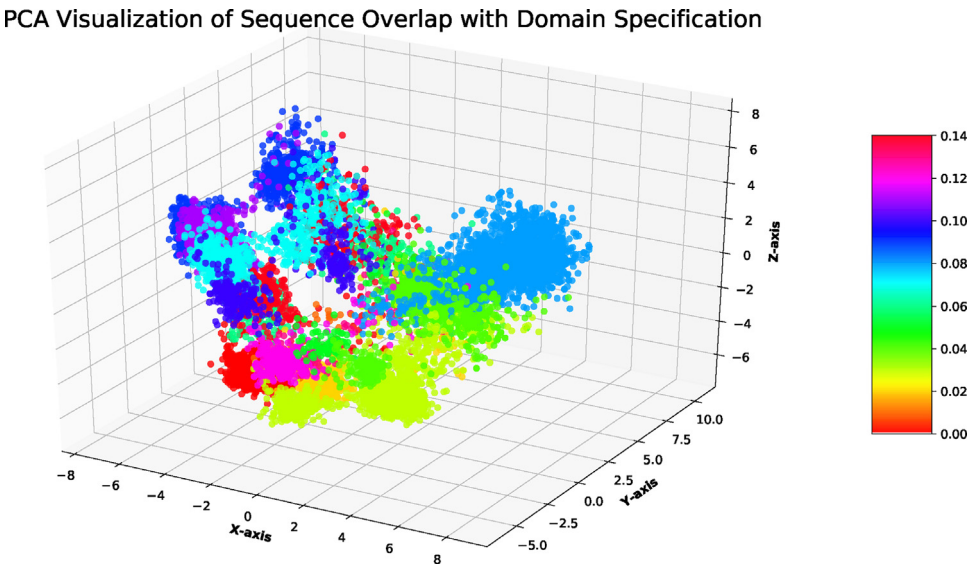


Fig. 9. PCA visualization of sequence overlap in service usage code snippets (with domain specification).

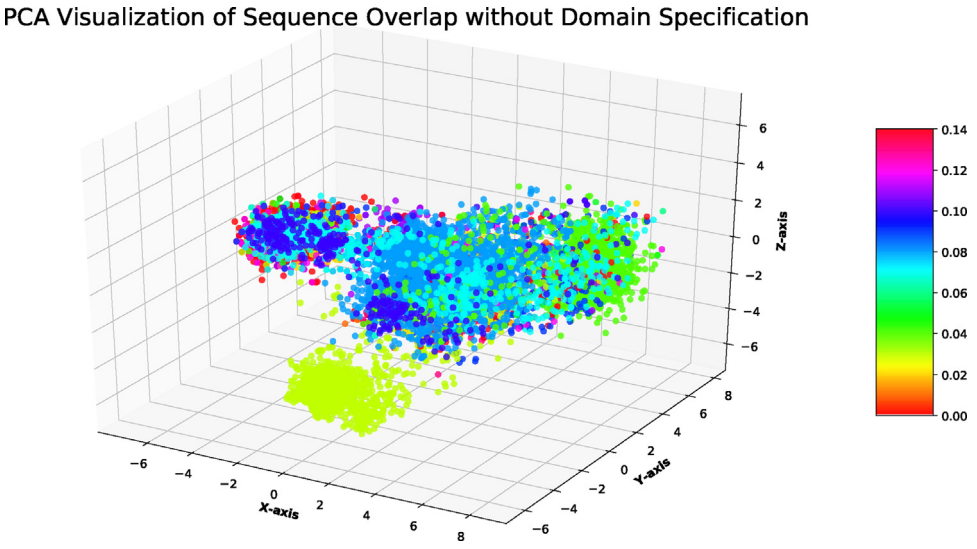


Fig. 10. PCA visualization of sequence overlap in service usage code snippets (without domain specification).

well-known libraries or web services calls. Therefore, auto-completion can put relevant codes in the list of suggested codes. The rank-aware evaluation metrics *e.g.,* Mean Reciprocal Rank (MRR) can demonstrate this option better. The mean reciprocal rank is a statistic calculation for evaluating models that returns a list, ordered by the probability of correctness (Radev et al., 2002). Therefore, we evaluate models using the de-noising strategy and random file selection by MRR on web service usages, in which the de-noising strategy achieved MRR = 56.23 and random file selection achieved MRR = 41.52. To this end, the model that uses the de-noising strategy cares more about the relevant code suggestion achieved by removing passive codes and feeding models with more relevant service usages.

4.3. Investigating impact of conditional code auto-completion

This section investigates the proposed method used for conditional code auto-completion and its impact on the performance of code auto-completion. As stated in Section 3.3, one of the challenges of code auto-completion is the similarity of web service endpoint URLs and their parameters. Figs. 9 and 10 demonstrate the similarity in web service usage with and without domain specification. As illustrated, the domain specification reduced the sequence overlapping. This plot uses PCA for visualizing code snippets sequence overlap.

In this context, this work tries to overcome this similarity and lack of a clear margin by using the domain on which the web service is implemented. In this way, domain-specified codes provide clear margins and help the learning algorithm's performance.

Fig. 11 shows the number of losses that occurred in the steps implemented in the learning phase. It is clear from this figure that the lowest loss event is related to *DCServCG*, which shows the positive effect of using the domain. The *DCServCG* model uses data with a specified domain, and the *ServCG* and *ServgCG_den* models use GPT-2 and data without a specified domain. Even when using DistilGPT2, which is a smaller model of GPT-2, *ServgCG_den* has lower training losses and higher accuracy.

To correctly evaluate a code auto-completion model, it is necessary to examine different aspects of the system's response. In fact, in addition to syntax-correct code prediction, it is required to examine factors such as context awareness and coherence of generated code. Therefore, the accuracy metric cannot demonstrate language modeling performance satisfactorily.

Table 7
Effectiveness of the Proposed de-noising strategy.

Data	OSS projects	Files	Passive classes	Active classes
Train codes	787	216 399	750 069	1 125 298
Selected codes/Label = 1	787	158 042	248 307	939 630
Noisy codes/Label = 0	787	58 357	501 762	185 668

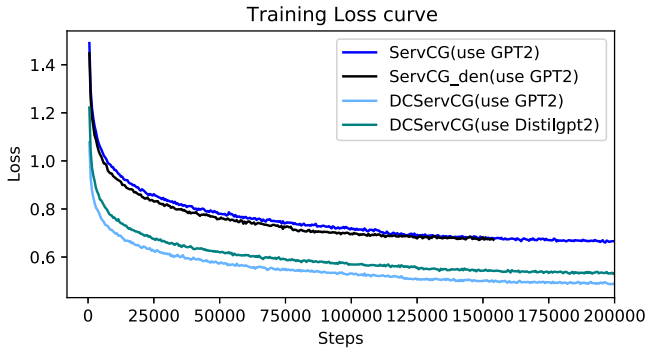


Fig. 11. Loss plot of trained models during training process.

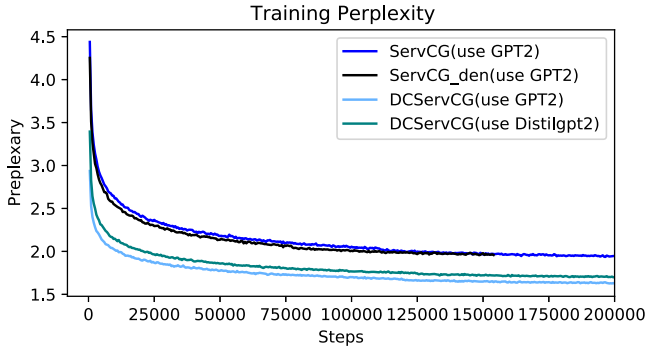


Fig. 12. Perplexity plot of trained models during training process.

Apart from human supervision for the quality of generated code which requires more effort in large-scale data, some metrics can automatically be obtained via mathematical or NLP concepts. For example, these metrics can consider the performance of the model on unseen data or calculate the word overlap based on the n-gram between the generated text and the test data. Perplexity is a well-known language modeling metric that is obtained by the test set's inverse probability, normalized by the number of sequences (Guo et al., 2021). So, if we have a tokenized sequence $X = (x_0, x_1, \ldots, x_t)$, then the Perplexity of the sequence of X can be calculated by Eq. (8):

$$PPL(X) = P(x_0, x_1, \ldots, x_t)^{-\frac{1}{t}} \tag{8}$$

$$PPL(X) = \sqrt[t]{\frac{1}{P(x_0, x_1, \ldots, x_t)}} \tag{9}$$

Therefore, a language model with acceptable performance is one that has high probability and smaller perplexity. Fig. 12 indicates the status of the Perplexity calculation on training data. The best result is for the *DCServCG* model with the smallest perplexity equal to 2.0959. It is worth mentioning that a smaller perplexity value represents a better auto-complete model performance. In addition, Fig. 15 shows the status of the Perplexity value calculated on the test data. The lowest value obtained for the *DCServCG* model is associated with the best result.

Figs. 11 and 12 show that the training loss curve and perplexity criteria are calculated and displayed to check the models' behavior during training. Moreover, this study employs generative models that

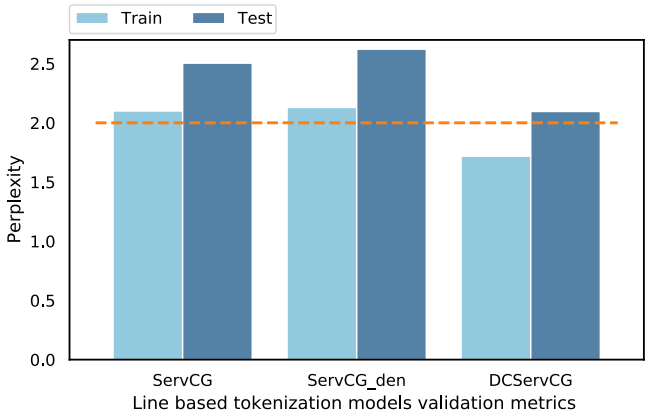


Fig. 13. Plot of train/test perplexity for trained models.

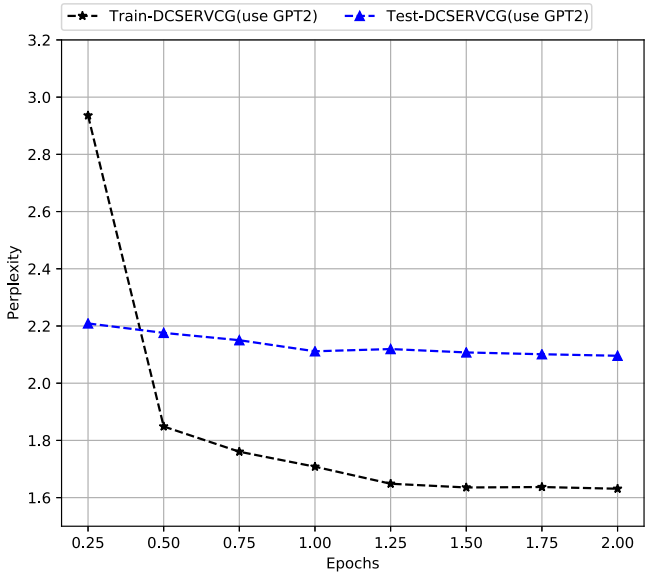


Fig. 14. Plot of train/test perplexity for DCSERVCG.

can also suffer from underfitting and overfitting challenges. Consequently, the proposed method tries to overcome these challenges in the data-gathering and training phases. In the data gathering phase, this study not only tries to cover all web services, including rare web service usage in training and test data but also provides conditional code generation, which offers more control over the attributes of the output text. In the training phase, adjusting model hyper-parameters and applying dropout increase model generalization. Figs. 13 and 14 demonstrate metric details to allow for easy comparison between the train/test perplexity metric.

According to the obtained results, using conditional text generation in the *DCServCG* (uses GPT-2) and *DCServCG* (uses DistilGPT2) models provide these models to learn better than other models. Based on Fig. 15, the *DCServCG* (uses GPT2) and *DCServCG* (uses DistilGPT2) models have better results than other models due to the high performance of conditional text generation. These results demonstrate the transformer-based ability to predict distinct types of tokens, which

Table 8
Experimental results on token prediction.

Model	Transformer	Tokenization	ACU	Perplexity	Train time	Model parameters
ServCG	GPT-2	Line	0.829	2.5038	16.34 h	124M
ServCG_den	GPT-2	Line	0.822	2.622	11.27 h	124M
DCServCG	GPT-2	Line	**0.862**	**2.0959**	15.3 h	124M
DCServCG	DistilGPT2	Line	0.858	2.1417	11.38 h	82M
CG_g	GPT-2	Line	0.7954	3.2209	21.24 h	124M
CG_den_g	GPT-2	Line	0.7808	3.4953	14.44 h	124M
ServCG	GPT-2	File	0.7008	6.3129	18.41 h	124M
ServCG_den	GPT-2	File	0.6942	6.5403	13.04 h	124M
DCServCG	GPT-2	File	0.6916	6.022	12.79 h	124M
DCServCG	DistilGPT2	File	0.6843	7.1092	10.82 h	82M

Table 9
Experimental results on perfect prediction.

Model	Token masked				
	i = 1	i = 2	i = 3	i = 4	i = 5
CG_g	93.319%	49.337%	24.434%	13.481%	3.815%
DCServCG	96.278%	68.124%	39.949%	22.069%	7.226%

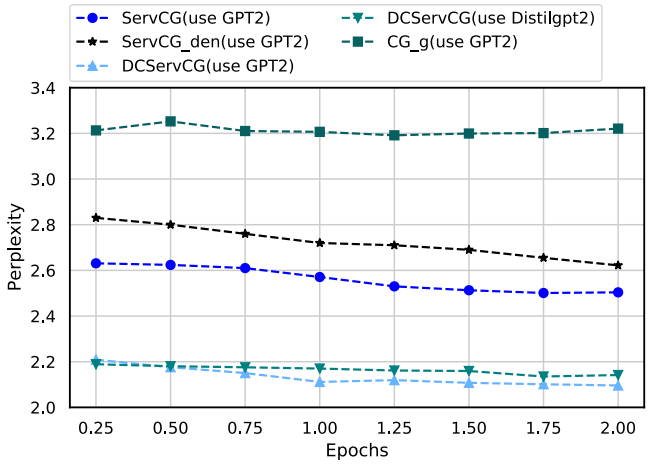


Fig. 15. Plot of test perplexity based on the number of training epochs.

is considered in *DCServCG* by annotating code by service domain. In addition, as shown in Table 8, the *DCServCG* performs better on split line-based compared to the file-based data. The reason is that transformers correctly reproduce labeled code lines; nonetheless, memorizing long texts such as labeled code files is challenging for the transformers *e.g.,* GPT-2. Moreover, the costs of the computational and memory resources grow considering the increase of the length.

Table 8 shows the experimental results of the training language models based on both line and file tokenization. In fact, perplexity value is mostly related to several factors, *e.g.,* perplexity results for character level will be better than token level, and token level consequently will have better results than line-based prediction. This concept is demonstrated in the table. Overall, based on automatic validation, *DCServCG*, which predicts tokens on a conditional annotated training dataset, shows smaller perplexity. The most remarkable outcome to emerge from the evaluation metrics is that conditional auto-completion helps to control output and provide a better result. Moreover, as illustrated in Table 8, there is a correlation between source-code tokenization and conditional auto-completion effectiveness. In the models that use file-based tokenization, the effect of labels is reduced with a large length on the left side of the input.

Another quality metric is called perfect prediction (Ciniselli et al., 2021a), which is more application-oriented. This metric is designed to calculate a prediction that matches programmers' coding in real-world projects. This study selected this metric since the generated code will be un-executable if a minor error is made in the generated code. Calculating perfect prediction requires a token masking strategy. Respecting this, the decoder-only transformers *e.g.,* GPT-2 take the previous tokens (left-side) as an input to predict the next tokens and generate codes left-to-right. Therefore, this study uses the right-side masked token strategy. In this context, if a tokenized sequence $X_{0-t} = (x_0, x_1, \ldots, x_t)$, i stands for number of iterations. The masked token is computed by the liner function $X_{n.i} = X_t - X_{t-n.i}$.

Table 9 indicates the perfect predictions with maximum iteration $n = 5$, in which *DCServCG* performs better than the general-purpose model, CG_g, trained by source code of the GitHub Java Corpus dataset. The model configuration is set equally for both models. Moreover, some parameters *e.g.,* temperature and top-k, can adjust the greedy search level of generative models. Therefore, the temperature was set to a lower value to decrease the greedy search effect, which tends to select the optimal global options rather than the optimal local selections.

4.4. More analysis of generator models coherency

In order to increase the accuracy and coherency of generator models, several configurations can be made during training and generation. One basic strategy for auto-generative language models LM is Greedy search decoding which is a quick algorithm that aims to select the code token with the highest probability (Ghosh et al., 2022); However, the significant problem is the lack of coherency. These problems led to generating non-executable codes which are more noticeable in the OSS big projects. Although these projects have high star and fork, they include excessive passive codes due to wrong DL training. This study addresses this problem using the noise reduction strategy (see code 3.2.1). The proposed de-noising strategy helps to reduce search space by filtering unusable codes. Moreover, parameters such as "top_k", "top_p" (Fan et al., 2018; Radford et al., 2019) are configured to generate more executable and relevant code snippets.

Although *ServgCG_den* model suggests that the most relevant codes via well configuration of model parameters, it cannot control the sequence overlapping in code snippets, which is more exaggerated in web service development 10. Table 10 shows code generation with and without conditional approach using GPT-2 transformer trained by the same dataset in which code generation started by the same token 'import'. The table demonstrates that the combination of the proposed de-noising strategy, conditional generation with a set of acceptable parameters provides significant improvements that decrease model confusion in sequence overlapping code snippets in web service usages.

4.5. Limitations of the study and future works

The limitations of the study can fall into two categories: (1) limited access to rare source codes and (2) computational resources. The first one refers to the challenge that the collected source codes come from online repositories, and the source codes have already been prepared. In addition, there are some online services that are not as popular. Therefore, such small communities make them difficult to find the codes they use. As a future work, other resources, such as StackOverflow, can be used. The second limitation is related to computational

Table 10
Comparison results of different conditional and unconditional approaches.

Approach	Search strategy	Line based Temperature = 0.6, top_k = 40, top_p = 0.95
ServCG	Greedy search	[–] import *com.axelor.apps.account.db.repo.InvoiceRepository;* [–] URL url = new URL("http*://www.google.com";*
	Beam search	[–] import *net.osmand;* [–] import *org.junit.Test;* [–] import *com.google.gson.stream.JsonReader;* [–] import *org.springframework.transaction.annotation.Transactional;*
ServgCG_den	Greedy search Beam search	[–] import *android.view.ViewGroup;* [–] import *javax.servlet;* [–] import *javax.xmlmapper;* [–] import *android.widget.LinearLayout;* [–] URL url = new URL("*http://localhost:" + getRpHttpsPort( );* [–] import *javax.servlet.http.HttpServletRequest;*
DCServCG	Greedy search	[VIDEO][YOUTUBE] import *android.util.DisplayMetrics;* [ADVERESTING][AMAZON] import *org.elasticsearch.common.SuppressForbidden;*
	Beam search	[ADVERESTING][AMAZON] *import java.security.KeyManager;* [MAP][GOOGLE] import *android.graphics.drawable.Drawable;* [VIDEO][YOUTUBE] URL url = new URL("http*://www.googleapis.com/youtube/v3/videos?part=<NUM_PARAM>* [SOCIAL][LINKEDIN] URL url = new URL("http*://api.linkedin.com/uas/oauth/authorize");* [MUSIC][LASTFM] URL url = new URL("http*://ws.audioscrobbler.com/2.0/")* [CALENDAR][GOOGLE] URL url = new URL("http*://api.calendar.com/");* [VIDEO][YOUTUBE] import *android.content.Context*

requirements and the size of transformers. We observe the GPT-2 with a large size performs well. However, current IDEs can include several AI-based extensions, for which using heavy models may be challenging. The DCServCG tool is a part of the SmartCLIDE[2] project, an service development online IDE, since SmartCLIDE, a cloud-based IDE, is associated with several functionalities and resource limitations for user sessions. Therefore, it requires a response to multiple users at once. In future work, we explore the *DCServCG* by employing it in the online SmartCLIDE toolkit and observing its behavior by considering the human judgment score. Moreover, most proposed code-completion models, including our study, provide output using GPT2 metrics such as temperature, top_k, and top_p to control output. Future works could be extended to ranking output based on features, such as the status of the creator of the repository, the number of forks, the stars, etc. This information can be embedded in the source line as we annotate the web service domain on the code line and filter while post-processing.

5. Conclusion

In SOA, developers tend to use and combine existing services to reduce costs and increase speed in the design and development process. The main challenge of this solution is the lack of a unique web service registry and consequently, fewer machine-readable web service technical documents. This study has attempted to compensate for the lack of technical documents by using the source codes of web service-based projects. In this way, the auto-complete model automatically helps the developers to complete the codes, including web service usage. Therefore, this work not only used the web service metadata for conditional auto-completion, but also considered public open-source projects that employ web services as a worthwhile resource for web service data extraction. For this purpose, first, a list of well-known and widely used services is extracted from ProgrammableWeb. On the basis of the extracted services, the source code of service-based projects that use these services are collected from GitHub. Afterward, the data-centric methods, including de-noising strategy and conditional code are applied to provide higher performance. The proposed approach provides a smaller perplexity (2.0959), which means a more accurate code auto-complete model. Moreover, *DCServCG* achieves a higher number of perfect predictions than the CG_g model, trained by the source code of the GitHub Java Corpus dataset. Experimental results show that conditional code compilation can provide large controllable token prediction in the web service domain with close concept and sequence overlap. Training time and resource usage are other critical characteristics affecting the performance of the auto-complete models using transformers in realistic environments. The proposed de-noising strategy in the data-centric approach has effectively reduced training time and resource usage. Using this strategy by removing 26.96% of noisy data led to a 31.02% improvement in training time with a minor effect on perplexity. It also significantly impacts the suggestion of reusable codes.

[2] SmartCLIDE https://smartclide.eu/.

CRediT authorship contribution statement

Zakieh Alizadehsani: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Hadi Ghaemi:** Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Visualization. **Amin Shahraki:** Supervision, Investigation, Data curation, Writing – original draft, Writing – review & editing. **Alfonso Gonzalez-Briones:** Funding acquisition, Project administration. **Juan M. Corchado:** Supervision, Funding acquisition, Project administration.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Zakieh Alizadesani reports financial support was provided by European Union. Zakieh Alizadehsani reports a relationship with European Union that includes: funding grants.

Data availability

Data will be made available on request.

Acknowledgments

This research has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No 871177 — This work has been supported by the Institute for Business Competitiveness of Castilla y León, and the European Regional Development Fund under grant CCTT3/20/SA/0002 (AIR-SCity project)

References

Akkiraju, R., Farrell, J., Miller, J.A., Nagarajan, M., Sheth, A.P., Verma, K., 2005. Web Service Semantics-WSDL-S. The Ohio Center of Excellence in Knowledge-Enabled Computing (Kno.e.sis).

Alizadehsani, Z., Feitosa, D., Maikantis, T., Ampatzoglou, A., Chatzigeorgiou, A., Berrocal, D., Briones, A.G., Corchado, J.M., Mateus, M., Groenewold, J., 2022. Service classification through machine learning: Aiding in the efficient identification of reusable assets in cloud application development. In: 2022 48th Euromicro Conference on Software Engineering and Advanced Applications. SEAA, IEEE, pp. 247–254.

Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C., 2018. A survey of machine learning for big code and naturalness. ACM Comput. Surv. 51 (4), 1–37.

Allamanis, M., Sutton, C., 2013a. Mining source code repositories at massive scale using language modeling. In: 2013 10th Working Conference on Mining Software Repositories. MSR, IEEE, pp. 207–216.

Allamanis, M., Sutton, C., 2013b. Mining source code repositories at massive scale using language modeling. In: The 10th Working Conference on Mining Software Repositories. IEEE, pp. 207–216.

Alsmadi, I., Aljaafari, N., Nazzal, M., AlHamed, S., Sawalmeh, A., Vizcarra, C.P., Khreishah, A., Anan, M., Algosaibi, A., Alnaeem, M., et al., 2022. Adversarial machine learning in text processing: A literature survey. IEEE Access.

Alwasouf, A.A., Kumar, D., 2019. Research challenges of web service composition. Softw. Eng. 681–689.

Bashath, S., Perera, N., Tripathi, S., Manjang, K., Dehmer, M., Streib, F.E., 2022. A data-centric review of deep transfer learning with applications to text data. Inform. Sci. 585, 498–528.

Belinkov, Y., Poliak, A., Shieber, S.M., Van Durme, B., Rush, A.M., 2019. Don't take the premise for granted: Mitigating artifacts in natural language inference. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, Florence, Italy, pp. 877–891. http://dx.doi.org/10.18653/V1/P19-1084, July 2019.

Chen, C., Peng, X., Xing, Z., Sun, J., Wang, X., Zhao, Y., Zhao, W., 2021. Holistic combination of structural and textual code information for context based api recommendation. IEEE Trans. Softw. Eng..

Cheng, H., Zhong, M., Wang, J., 2020. Diversified keyword search based web service composition. J. Syst. Softw. 163, 110540.

Church, K.W., 2017. Word2Vec. Nat. Lang. Eng. 23 (1), 155–162.

Ciniselli, M., Cooper, N., Pascarella, L., Mastropaolo, A., Aghajani, E., Poshyvanyk, D., Di Penta, M., Bavota, G., 2021a. An empirical study on the usage of transformer models for code completion. IEEE Trans. Softw. Eng..

Ciniselli, M., Cooper, N., Pascarella, L., Poshyvanyk, D., Di Penta, M., Bavota, G., 2021b. An empirical study on the usage of BERT models for code completion. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories. MSR, IEEE, pp. 108–119.

Deng, S., Huang, L., Taheri, J., Yin, J., Zhou, M., Zomaya, A.Y., 2016. Mobility-aware service composition in mobile communities. IEEE Trans. Syst. Man Cybern.: Syst. 47 (3), 555–568.

Fan, A., Lewis, M., Dauphin, Y., 2018. Hierarchical neural story generation. arXiv preprint arXiv:1805.04833.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Gamma, E., Helm, R., Johnson, R., Johnson, R.E., Vlissides, J., et al., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Deutschland GmbH.

Gharineiat, A., Bouguettaya, A., Ba-hutair, M.N., 2021. A deep reinforcement learning approach for composing moving IoT services. IEEE Trans. Serv. Comput..

Ghosh, I., Ivler, M., Ramamurthy, S.R., Roy, N., 2022. SpecTextor: End-to-end attention-based mechanism for dense text generation in sports journalism. In: 2022 IEEE International Conference on Smart Computing. SMARTCOMP, IEEE, pp. 362–367.

Gu, X., Zhang, H., Zhang, D., Kim, S., 2016. Deep API learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 631–642.

Guo, B., Wang, H., Ding, Y., Wu, W., Hao, S., Sun, Y., Yu, Z., 2021. Conditional text generation for harmonious human-machine interaction. ACM Trans. Intell. Syst. Technol. 12 (2), 1–50.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., Brockschmidt, M., 2019. Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436.

Izadi, M., Gismondi, R., Gousios, G., 2022. CodeFill: Multi-token code completion by jointly learning from structure and naming sequences. In: Proceedings - 2022 ACM/IEEE 44th International Conference on Software Engineering, ICSE 2022. In: Proceedings - International Conference on Software Engineering, IEEE, United States, pp. 401–412. http://dx.doi.org/10.1145/3510003.3510172.

2019. Javalang, Provies a lexer and parser. https://pypi.org/project/javalang/, [Online; accessed 2022].

Levenshtein, V.I., et al., 1966. Binary codes capable of correcting deletions, insertions, and reversals. In: Soviet Physics Doklady, Vol. 10. Soviet Union, pp. 707–710.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., Zettlemoyer, L., 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461.

Li, S., Luo, H., Zhao, G., Tang, M., Liu, X., 2022. Bi-directional Bayesian probabilistic model based hybrid grained semantic matchmaking for web service discovery. World Wide Web 25 (2), 445–470.

Liu, M., Tu, Z., Zhu, Y., Xu, X., Wang, Z., Sheng, Q.Z., 2021. Data correction and evolution analysis of the ProgrammableWeb service ecosystem. J. Syst. Softw. 182, 111066.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C.B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S.K., Fu, S., Liu, S., 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. CoRR abs/2102.04664.

Lydia, A., Francis, S., 2019. Adagrad—an optimizer for stochastic gradient descent. Int. J. Inf. Comput. Sci 6 (5), 566–568.

Ma, S.-P., Chen, Y.-J., Syu, Y., Lin, H.-J., Fanjiang, Y.-Y., 2018. TEST-Oriented RESTful service discovery with semantic interface compatibility. IEEE Trans. Serv. Comput. 14 (5), 1571–1584.

Ma, S.-P., Chen, Y.-J., Syu, Y., Lin, H.-J., Fanjiang, Y.-Y., 2021a. Test-oriented restful service discovery with semantic interface compatibility. IEEE Trans. Serv. Comput..

Ma, T., Pan, Q., Rong, H., Qian, Y., Tian, Y., Al-Nabhan, N., 2021b. T-bertsum: Topic-aware text summarization based on bert. IEEE Trans. Comput. Soc. Syst. 9 (3), 879–890.

Martin, R.C., Grenning, J., Brown, S., Henney, K., Gorman, J., 2018. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.

Mirjalili, S., Lewis, A., 2016. The whale optimization algorithm. Adv. Eng. Softw. 95, 51–67.

2018. OPENAI, AI research and deployment company. https://beta.openai.com/docs/guides/, [Online; accessed 2022].

Pan, S.J., Yang, Q., 2009. A survey on transfer learning. IEEE Trans. Knowl. Data Eng. 22 (10), 1345–1359.

Papineni, K., Roukos, S., Ward, T., Zhu, W.-J., 2002. Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. pp. 311–318.

2014. Programmableweb, Dataset. [16] https://www.programmableweb.com/api/, [Online; accessed 2022].

Radev, D.R., Qi, H., Wu, H., Fan, W., 2002. Evaluating web-based question answering systems. In: LREC. Citeseer.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al., 2019. Language models are unsupervised multitask learners. OpenAI Blog 1 (8), 9.

Rafsanjani, M.K., Rezaei, A., Shahraki, A., Saeid, A.B., 2014. QARIMA: A new approach to prediction in queue theory. Appl. Math. Comput. 244, 514–525.

Rahman, M., Watanobe, Y., Nakamura, K., et al., 2020. A neural network based intelligent support model for program code completion. Sci. Program. 2020.

Raychev, V., Bielik, P., Vechev, M., 2016. Probabilistic model for code with decision trees. ACM SIGPLAN Not. 51 (10), 731–747.

Rochwerger, B., Breitgand, D., Levy, E., Galis, A., Nagin, K., Llorente, I.M., Montero, R., Wolfsthal, Y., Elmroth, E., Caceres, J., et al., 2009. The reservoir model and architecture for open federated cloud computing. IBM J. Res. Dev. 53 (4), 4–1.

Sangaiah, A.K., Bian, G.-B., Bozorgi, S.M., Suraki, M.Y., Hosseinabadi, A.A.R., Shareh, M.B., 2020. A novel quality-of-service-aware web services composition using biogeography-based optimization algorithm. Soft Comput. 24 (11), 8125–8137.

Schmidt, D.C., 2006. Model-driven engineering. Comput.-IEEE Comput. Soc.- 39 (2), 25.

Seltman, H.J., 2012. Experimental design and analysis.

Shahraki, A., Geitle, M., Haugen, Ø., 2020. A comparative node evaluation model for highly heterogeneous massive-scale Internet of Things-Mist networks. Trans. Emerg. Telecommun. Technol. 31 (12), e3924.

Shahraki, A., Haugen, Ø., 2019. An outlier detection method to improve gathered datasets for network behavior analysis in IoT. J. Commun..

Sun, X., Wang, S., Xia, Y., Zheng, W., 2019. Predictive-trend-aware composition of web services with time-varying quality-of-service. IEEE Access 8, 1910–1921.

Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N., 2020. Intellicode compose: Code generation using transformer. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1433–1443.

Syriani, E., Luhunu, L., Sahraoui, H., 2018. Systematic mapping study of template-based code generation. Comput. Lang. Syst. Struct. 52, 43–62.

2019. TabNine, Autocompletion with deep learning. https://www.tabnine.com, [Online; accessed 2022].

Tang, B., Yan, M., Zhang, N., Xu, L., Zhang, X., Ren, H., 2021. Co-attentive representation learning for web services classification. Expert Syst. Appl. 180, 115070.

Terada, K., Watanobe, Y., 2019. Code completion for programming education based on recurrent neural network. In: 2019 IEEE 11th International Workshop on Computational Intelligence and Applications. IWCIA, IEEE, pp. 109–114.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.

Wang, S., Liu, J., Qiu, Y., Ma, Z., Liu, J., Wu, Z., 2019. Deep learning based code completion models for programming codes. In: Proceedings of the 2019 3rd International Symposium on Computer Science and Intelligent Control, pp. 1–9.

Wang, X., Sun, Q., Liang, J., 2020. Json-ld based web api semantic annotation considering distributed knowledge. IEEE Access 8, 197203–197221.

Wang, X., Xu, H., Wang, X., Xu, X., Wang, Z., 2022. A graph neural network and pointer network-based approach for QoS-aware service composition. IEEE Trans. Serv. Comput..

Wu, S., Shen, S., Xu, X., Chen, Y., Zhou, X., Liu, D., Xue, X., Qi, L., 2022. Popularity-aware and diverse web APIs recommendation based on correlation graph. IEEE Trans. Comput. Soc. Syst..

Xiong, R., Wang, J., Zhang, N., Ma, Y., 2018. Deep hybrid collaborative filtering for web service recommendation. Expert Syst. Appl. 110, 191–205.

Xu, X., Chen, H., Moreno-Noguer, F., Jeni, L.A., De la Torre, F., 2021. 3D human pose, shape and texture from low-resolution images and videos. IEEE Trans. Pattern Anal. Mach. Intell..

Yang, Y., Qamar, N., Liu, P., Grolinger, K., Wang, W., Li, Z., Liao, Z., 2020. Servenet: A deep neural network for web services classification. In: 2020 IEEE International Conference on Web Services. ICWS, IEEE, pp. 168–175.

Zhang, N., Wang, J., Ma, Y., He, K., Li, Z., Liu, X.F., 2018. Web service discovery based on goal-oriented query expansion. J. Syst. Softw. 142, 73–91.

Zhou, W., Kim, S., Murali, V., Aye, G.A., 2022. Improving code autocompletion with transfer learning. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, pp. 161–162.

Zou, G., Yang, S., Duan, S., Zhang, B., Gan, Y., Chen, Y., 2022. DeepLTSC: Long-tail service classification via integrating category attentive deep neural network and feature augmentation. IEEE Trans. Netw. Serv. Manag..