



MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR  
DE LA RECHERCHE SCIENTIFIQUE ET DE L'INNOVATION  
UNIVERSITÉ SULTAN MOULAY SLIMANE  
ÉCOLE NATIONALE DES SCIENCES APPLIQUÉS  
DE KHOURIBGA



# ORACLE

## End of Studies Report

for obtaining the

Engineering Degree Diploma

Major : Computer Science and Data Engineering



Presented by :  
Yasser Zarhloul

## Towards faster development with AI-enabled applications

Under the supervision of :  
Ms. Nidal **Lamghari**, *ENSA Khouribga*  
Mr. Jonas **Schweizer**, *Oracle*  
Mr. Damien **Hilloulin**, *Oracle*

Année Académique : 2023 - 2024



# Summary

<b>Dedication</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>Résumé</b>	<b>iv</b>
<b>Abstract</b>	<b>v</b>
<b>Contents</b>	<b>vii</b>
<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xi</b>
<b>Acronyms</b>	<b>xii</b>
<b>General Introduction</b>	<b>1</b>
<b>1 General Context of the Internship</b>	<b>3</b>
<b>2 Evaluating and Improving Prompts for Choice Selection</b>	<b>12</b>
<b>3 Code Search for License Compliance</b>	<b>28</b>
<b>General Conclusion</b>	<b>48</b>
<b>Bibliography</b>	<b>49</b>
<b>Bibliography</b>	<b>49</b>

# Dedication

“

*To my family and friends, this thesis reflects all the love, support, and encouragement you've given me.*

*To my family, thank you for always being there when I needed advice or a push to keep going. You have helped me chase my dreams.*

*To my friends, thank you for being with me during the hard and the good times. Your friendship and trust in me have meant a lot. This success is as much yours as it is mine.*

*Finally, to myself, in the words of George Hotz: "we do things because we can."*

”

**- Yasser**

# Acknowledgements

I would like to express my gratitude to all those who have contributed, directly or indirectly, to the successful completion of my final year project.

I begin by thanking my academic supervisor, Ms. Nidal Lamghari, and my mentor Mr. Jonas Schweizer for their constructive instructions, valuable encouragement, and insightful advice. I appreciate their guidance and monitoring throughout the internship period.

I am grateful to all the staff at Oracle MADC for their support and their generosity in sharing and transferring skills.

I also wish to express my gratitude to my project manager, Mr. Damien Hilloulin, as well as to the entire team for their cooperation and a big thank you to my local manager, Ms. Fatima Zahra Mouak, for her assistance and counsel.

Then, many thanks to all the staff at ENSA and the juries. Your diligence have greatly helped throughout our journey. Your expertise is truly appreciated.

Finally, I would like to sincerely thank my classmates for their camaraderie, sharing, and all the enjoyable moments we spent together.

# Résumé

Le développement des assistants personnels utilisant des grands modèles de langage, représente une avancée technologique significative, augmentant la productivité dans le développement logiciel et allégeant la charge de travail manuelle. Cependant, l'utilisation de ces outils AI soulève des défis importants liés à la propriété intellectuelle et à la gestion des licences, car ces modèles sont formés sur des ensembles de données contenant du code sous licences restrictives. Ce qui peut exposer les entreprises à des risques légaux.

Pour limiter ces risques, il est essentiel que les entreprises mettent en place des technologies avancées de détection de code pour identifier les segments problématiques et vérifier la conformité des licences avant le déploiement du code. Cela nécessite l'intégration d'outils de recherche de code capables de distinguer les subtilités des licences logicielles. En intégrant ces mesures de sécurité et en renforçant la surveillance, les entreprises peuvent pleinement bénéficier des avantages des modèles tout en se conformant aux cadres juridiques et en évitant les problèmes légaux.

---

**Keywords :** Assistants personnels, Grand Modèle de Langage, Productivité, Développement de logiciels, Génération de code, Risque de licence, Conformité légale, Balayage de code, Recherche de code

---

# Abstract

The development and use of personal assistants powered by Large Language Models (LLMs) are transforming technology, enhancing productivity for developers and general users alike. These AI tools efficiently assist in tasks like software development by automating code generation, which speeds up projects and reduces manual work.

However, these capabilities carry risks related to intellectual property and licensing because LLMs train on large datasets, including code under restrictive licenses. Using such code could expose companies to legal issues, including license violations with serious repercussions.

To address these concerns, it is crucial for companies to use advanced scanning technologies to detect problematic code generated by LLMs. These tools, integrated within the software development lifecycle, must understand license nuances and identify similarities between generated and existing code to ensure compliance before deployment.

This approach underlines the dual benefits and challenges of integrating sophisticated AI into everyday tools. It necessitates enhanced oversight and control to align with legal standards in software development, ensuring that companies can capitalize on the efficiency of AI while safeguarding against legal risks.

---

**Keywords :** Personal Assistants, Large Language Models, Productivity, Software Development, Code Generation, Licensing Risk, Legal Compliance, Code Scanning, Code Search

---

# ملخص

تمثل تطوير وتنفيذ المساعدين الشخصيين باستخدام النماذج اللغوية العملاقة، تقدمًا تكنولوجيًا مهمًا يهدف إلى زيادة الإنتاجية في تطوير البرمجيات وتخفيف الأعباء اليدوية. ومع ذلك، يرتبط استخدام هذه الأدوات الذكية بتحديات كبيرة متعلقة بالملكية الفكرية وإدارة التراخيص، حيث أن هذه النماذج مدربة على مجموعات بيانات تحتوي على كود خاضع لتراخيص مقيدة، مما قد يعرض الشركات لمخاطر قانونية. للحد من هذه المخاطر، من الضروري أن تطبق الشركات تقنيات متقدمة لكشف الكود لتحديد الأجزاء المشككة والتحقق من التزام التراخيص قبل نشر الكود. هذا يتطلب دمج أدوات بحث الكود قادرة على التمييز بين دقائق التراخيص البرمجية. بدمج هذه التدابير الأمنية وتعزيز الرقابة، يمكن للشركات الاستفادة الكاملة من مزايا النماذج اللغوية العملاقة مع الالتزام بالإطار القانوني وتجنب المشاكل القانونية.

---

**كلمات مفتاحية :** نماذج لغوية عملاقة، تطوير البرمجيات، زيادة الإنتاجية، الملكية الفكرية، إدارة التراخيص، كشف الكود، بحث الكود

---

# Contents

<b>Dedication</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>Résumé</b>	<b>iv</b>
<b>Abstract</b>	<b>v</b>
<b>Contents</b>	<b>vii</b>
<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xi</b>
<b>Acronyms</b>	<b>xii</b>
<b>General Introduction</b>	<b>1</b>
<b>1 General Context of the Internship</b>	<b>3</b>
Introduction . . . . .	3
1.1 Presentation of the host organization . . . . .	3
1.1.1 Oracle Corporation . . . . .	4
1.1.1.1 History . . . . .	4
1.1.1.2 Oracle MADC . . . . .	4
1.1.1.3 Oracle Labs . . . . .	5
1.1.1.4 Business Area of Activity . . . . .	6
1.1.2 Organizational structure . . . . .	6
1.1.3 Organization culture . . . . .	7
1.2 Project Presentation . . . . .	7
1.2.1 Objectives . . . . .	7
1.2.2 Proposed solutions . . . . .	8
1.3 Project Coordination . . . . .	9
1.3.1 Scrum Methodology . . . . .	9
1.3.2 Operational Tools . . . . .	9
Conclusion . . . . .	11
<b>2 Evaluating and Improving Prompts for Choice Selection</b>	<b>12</b>

Introduction	12
2.1 Project Context	12
2.1.1 General Objective	12
2.1.2 Competitive Analysis of Langchain	13
2.1.3 Assistant Capabilities	13
2.1.4 Main Components of the Assistant	14
2.1.5 Pre-defined Steps for the Assistant	14
2.1.6 Addressing Operational Challenges	14
2.1.6.1 Security	14
2.1.6.2 Reliability	15
2.1.6.3 Safety	15
2.2 Benchmark Design	15
2.2.1 Problem Statement	15
2.2.2 Competitive Analysis	15
2.2.3 Proposed Solution	16
2.2.4 Prompt Benchmark	16
2.3 Methodology	16
2.3.1 Ground Truth Dataset	17
2.3.1.1 Dataset Overview	17
2.3.1.2 Data Preprocessing	17
2.3.2 Prompt Format	18
2.3.3 Benchmarking prompts for choice selection	19
2.4 Experimental Setup	20
2.5 Experiments and Results	21
2.5.1 Sensitivity to user input	21
2.5.2 Using a short prompt template	21
2.5.3 Using Chain-of-Thought prompt template	23
2.5.4 Performance according to different prompt templates	24
2.5.5 Summary	25
2.5.6 Future Work	26
Conclusion	26
<b>3 Code Search for License Compliance</b>	<b>28</b>
Introduction	28
3.1 Project Presentation	28
3.1.1 Proposal	28
3.1.2 Problem Statements	28
3.1.3 Competitive Landscape	30
3.1.4 Proposed Solution	30
3.2 Literature Review of Code Clone Detection	31
3.2.1 Program Similarity	31
3.2.2 Source Code similarity	31

3.2.2.1	Code Clones Definition . . . . .	32
3.2.2.2	Code Similarity . . . . .	32
3.2.2.3	Granularity of Code Snippets . . . . .	32
3.2.2.4	Code Clone Terminology . . . . .	32
3.2.3	Code similarity detection methods . . . . .	33
3.2.3.1	Metric-based methods . . . . .	33
3.2.3.2	Text-based methods . . . . .	34
3.2.3.3	Token-based methods . . . . .	35
3.2.3.4	Tree-based methods . . . . .	35
3.2.3.5	Learning-based methods . . . . .	36
3.2.3.6	Hybrid methods . . . . .	36
3.2.4	Embeddings for Code Clone Detection . . . . .	36
3.2.4.1	Word Embeddings . . . . .	37
3.2.4.2	Code Embeddings . . . . .	37
3.2.4.3	Embedding-based methods for code clone detection . . . . .	38
3.3	Methodology . . . . .	39
3.3.1	Code similarity detection phases . . . . .	39
3.3.2	Code Similarity Evaluation . . . . .	40
3.3.2.1	Efficacy . . . . .	41
3.3.2.2	Search Efficiency . . . . .	41
3.3.2.3	Search Scalability and Execution Time . . . . .	41
3.3.3	Targeting Textual Similarity Types . . . . .	42
3.3.4	Targetting semantical similarity types . . . . .	44
3.3.4.1	Code Search for Retrieval . . . . .	44
3.3.4.2	Classification for Code Clone Matching . . . . .	44
3.3.5	Reference State-of-the-Art Results . . . . .	46
Conclusion	. . . . .	46
<b>General Conclusion</b>		<b>48</b>
<b>Bibliography</b>		<b>49</b>
<b>Bibliography</b>		<b>49</b>

# List of Figures

1.1	Oracle Logo . . . . .	3
1.2	Oracle MADC Location . . . . .	4
1.3	MADC Major milestones . . . . .	5
1.4	Executive Leadership Board . . . . .	6
2.1	Prompt accuracy with increasing complexity . . . . .	26
3.1	Word2Vec embeddings for documents X, Y . . . . .	37
3.2	CodeBERT training using the Masked Language Modeling objective . . . . .	38
3.3	GraphCodeBERT training using the Masked Language Modeling objective . . . . .	39
3.4	Pretraining tasks for CodeT5 . . . . .	39
3.5	Two-phase Methodology . . . . .	45
3.6	Detailed Example Walkthrough . . . . .	45
3.7	Results on natural language code retrieval. [14] . . . . .	46

# List of Tables

2.1	Benchmark Design User Story . . . . .	15
2.2	Examples of pairs of ill-formed and well-formed questions from the MQR dataset. . .	17
2.3	Elements of <i>BASELINE_PT</i> . . . . .	21
2.4	Accuracy according to different quality of user input . . . . .	21
2.5	Elements of <i>SHORT_PT</i> . . . . .	22
2.6	Summary of <i>SHORT_PT on 8_catg_set</i> . . . . .	22
2.7	Elements of <i>COT1SHOT_PT</i> . . . . .	23
2.8	Summary of <i>COT1SHOT_PT on 8_catg_set</i> . . . . .	24
2.9	Performance Metrics Across Different Categories . . . . .	25
3.1	Common metrics used for clone detection . . . . .	34
3.2	Transformations and Intermediate States . . . . .	40
3.3	Search-Related Features in OpenSearch . . . . .	42
3.4	Selected Query Types . . . . .	43
3.5	Recall and Precision on Clone Type Reasoning [11] . . . . .	46
3.6	Methods Recall and Precision [11] . . . . .	46

# Acronyms

AI	Artificial Intelligence
ML	Machine Learning
LLM	Large Language Model
MQR	Multi-domain Question Rewriting
QA	Question Answering
CUDA	Compute Unified Device Architecture
LCS	Longest Common Subsequence
TXL	Turing eXtender Language
AST	Abstract Syntax Tree
PDG	Program Dependency Graph
NN	Neural Networks
BERT	Bidirectional Encoder Representations from Transformers
MLM	Masked Language Modeling



# General Introduction

Drawing inspiration from human behavior, the concept of utilizing a collection of autonomous and distributed entities, known as agents, has emerged. These agents are individual software entities with a well-defined and limited scope, designed to perceive their environment and determine actions accordingly. In the realm of multi-agent-based approaches, these agents collaborate to address complex and emergent phenomena inspired by human interactions, such as expert collaboration within organizations to resolve customer requests. The architecture of such systems adheres to the "divide and conquer" paradigm, involving requester, provider, and middle agents that mediate responses to user requests through a structured process of capability advertisement, request mediation, and result composition.

In the contemporary digital landscape, chatbots have become increasingly prominent across various domains. However, designing a versatile chatbot capable of providing reasonable answers remains a formidable challenge. Current chatbot systems face significant limitations, including restricted scope and inadequacies in understanding user intent, task planning, tool utilization, and personal data management. These deficiencies have driven the focus toward task-specific assistants, aiming to enhance user efficiency in obtaining information and executing tasks, thereby offering a more intelligent, convenient, and enriched interaction experience.

Evaluating these assistants can be particularly challenging. Benchmarking and assessment processes often involve complex procedures that can hinder regular evaluations, making it difficult to enhance existing templates efficiently. Simplifying and automating these evaluation processes would encourage more frequent benchmarking and facilitate improvements in chatbot systems.

Assistants are not limited to conversational tasks; they also extend to code generation. The advent of generative AI tools, such as OpenAI's ChatGPT and GitHub Copilot, has introduced new complexities in software development. These tools, while facilitating the generation of code snippets, pose significant legal challenges due to the potential inclusion of code with restrictive licenses. This has underscored the need for robust mechanisms to identify and manage licensing risks associated with AI-generated code, ensuring compliance with intellectual property laws.

Code similarity detection remains a central focus in software engineering, particularly in man-

aging code quality and mitigating licensing conflicts. The presence of several large-scale online code databases, such as GitHub, poses significant challenges to code similarity detection. Traditional methods, which rely on pairwise comparisons of code fragments or tree-based detection tools, struggle with scalability. Consequently, there is a pressing need for scalable tools capable of efficiently identifying code clones across extensive databases, safeguarding software products against potential legal risks.

Addressing the code search problem is essential for ensuring that all AI-generated code within software products and systems is legally compliant and adheres to licensing requirements. This commitment to software integrity and compliance safeguards products against potential legal risks. The code search problem involves identifying suitable code within large codebases or across various software projects that meet specific requirements or queries. The significance of this research is rooted in two primary challenges: scalability of search systems and semantical alignment. Developing a scalable approach that accurately locates similar code fragments while avoiding expensive comparisons is crucial for effective code similarity detection.

# General Context of the Internship

## Introduction

This chapter provides an overview of the host organization, Oracle Corporation, detailing its history, key divisions, and areas of business activity. It also explores the organizational structure and culture within Oracle. Following this, the section presents the project undertaken during the thesis, outlining its objectives and proposed solutions. Finally, it discusses the project coordination approach, highlighting the use of Scrum methodology and operational tools that facilitate effective project management and team collaboration .

## 1.1 Presentation of the host organization



Figure 1.1: Oracle Logo

In today's world, businesses, technology companies, and the public sector rely on databases to store their data, while utilizing cloud services to manage their IT infrastructure and other essential tools such as ERP, CRM, SCM, EPM, and SPM. These technologies have taken a crucial role, making it inconceivable to function without them.

Oracle, a leading US multinational computer technology corporation, not only employs these tools in its day-to-day operations but also creates and provides them to customers. Moreover, the company places a high value on innovation and ensuring customer satisfaction.

The first section of this project provides a broad context by giving an overview of Oracle Corporation and Oracle Morocco Research Development Center (MADC). This includes an understanding of their business area of activity, their mission, as well as their organizational culture, and structure.

## 1.1.1 Oracle Corporation

### 1.1.1.1 History

Oracle Corporation is a Tech company with headquarters in Austin, Texas. In 1977, Larry Ellison, Bob Miner, and Ed Oates founded Oracle Corporation, then called as Software Development Laboratories (SDL). The company first focused on creating Oracle Database, a relational database management system (RDBMS), that became its flagship product. Enterprise software, cloud computing, hardware systems, and consulting services are now all part of Oracle's growing list of services and product options. Sun Microsystems was a key acquisition in 2010, giving Oracle access to technologies such as Java and the Solaris operating system.

Today, Oracle is a leading provider of cloud-based IT infrastructure and software that helps organizations grow, find new sources of efficiency, and improve their performance. To help organize and secure client data, Oracle developed the first and only autonomous database in the world as well as Oracle cloud.

### 1.1.1.2 Oracle MADC



Figure 1.2: Oracle MADC Location

The first Oracle RD center in Africa and one of only a few in the world, the Oracle MADC Morocco Research and Development Center, is a software development center established by Oracle Corporation in Casablanca, Morocco. The center was inaugurated in 2015 and is one of Oracle's key development centers in the EMEA region. The center is staffed by highly skilled and talented software developers, engineers, and project managers who work on a variety of projects across various

Oracle product lines utilizing the newest trends in innovation, such as artificial intelligence, augmented/virtual/mixed reality, big data analytics, Blockchain, cloud computing, cybersecurity, internet of things, machine learning, mobile computing, serverless computing. The center's researchers make use of all these technologies to tackle the most pressing problems facing business, science, and the public sector.

Expansion of internship and graduate recruitment initiatives and joint research projects with regional universities are all be part of this global program.

Oracle MADC has been successful in delivering several high-quality products and solutions, such as Oracle Cloud Infrastructure (OCI), Autonomous Data Warehouse (ADW), etc.

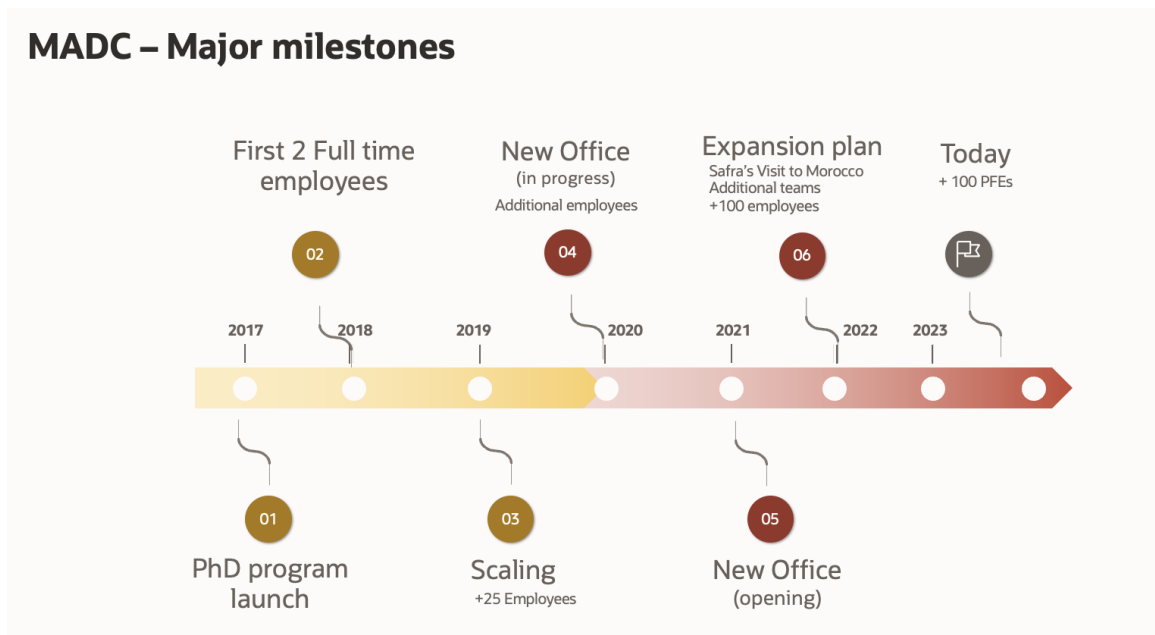


Figure 1.3: MADC Major milestones

### 1.1.1.3 Oracle Labs

The research and development division of Oracle is called Oracle Labs. which focuses primarily on creating technologies that will keep Oracle at the forefront of the IT industry. Researchers at Oracle Labs pursue innovative methodologies and approaches, frequently taking on challenges or projects that would be difficult to complete within a product development organization or that involved high risk or uncertainty. Real-world applications are the focus of Oracle Labs research, and the researchers are working to create technologies that will eventually have a big impact on how society and technology evolve. For instance, work done in Oracle Labs led to the development of chip multithreading and the Java computer language.

Oracle's quest for research and development is greatly supported by Oracle Labs. Oracle can recognize challenges, find creative solutions, and create new products that can spur growth and innovation in the future because of its well-rounded research portfolio.

### 1.1.1.4 Business Area of Activity

Oracle provides a variety of goods and services to its clients. Software development tools, cloud services, training, consultancy, and credentials are just a few of the goods and services offered here. One of the brands Oracle offers is Oracle Database, which has held the top spot since 1979 and is supported by cloud artificial intelligence. SaaS, PaaS, IaaS, and DaaS are just a few of the integrated solutions offered by Oracle Cloud Infrastructure for business, IT, and development needs. Oracle Middleware is also a platform that is integrated for creating and running intelligent, agile applications while increasing technical efficiency using contemporary software designs. Oracle Services, which include Oracle Advanced Customer Support Services, Oracle Premium Support, Oracle Consulting, Oracle Financing, and Oracle Managed Cloud Services.

### 1.1.2 Organizational structure

Oracle Corporation has a hierarchical organizational structure with three main business divisions: Hardware, Services, Cloud and License. Each segment is divided into smaller business units focused on goods, services, or geographical areas. The Board of Directors oversees the company's performance and strategic direction, while the Executive Leadership Team manages daily operations. Functional divisions, including sales, marketing, finance, human resources, legal, and IT, fall under the Executive Leadership Team. Business units are arranged by region, product, and service, with a senior VP or general manager overseeing each unit's operations. The employees are subordinated to both a functional manager and a business unit manager in the matrix-based organizational structure of the business units.



Figure 1.4: Executive Leadership Board

In general, Oracle's organizational structure is built to encourage teamwork, creativity, and client attention. While the matrix form enables people to work across many functions and business di-

visions to achieve shared goals, the hierarchical structure promotes effective decision-making and communication.

### 1.1.3 Organization culture

Oracle has a strong corporate culture that prioritizes innovation, client focus, and excellence. The company's history of innovation, emphasis on customer satisfaction, and dedication to fostering a friendly and collaborative work environment all have a bearing on the company's culture.

Oracle's culture places a strong emphasis on the following:

- **Innovation:** which is one of its defining characteristics. The business has a lengthy history of creating cutting-edge goods and technology that have revolutionized the tech sector. This culture of innovation is promoted by a dedication to research and development as well as by giving staff members the tools and encouragement they require to test out fresh concepts and create new products.
- **Customer Satisfaction:** Oracle's customer-centric approach to business reflects its commitment to providing goods and services that satisfy the needs of its clients. Employees are encouraged to collaborate closely with clients to comprehend their demands and create solutions that meet their problems.
- **Quality:** Oracle is dedicated to providing quality operations, products, and services, reflected in its solid reputation for quality, dependability, and performance. The company expects every person to strive for excellence, providing them with the tools, resources, and encouragement to succeed.
- **Collaboration and Teamwork:** Effective collaboration is essential for achieving Oracle's goals and delivering value to customers. The company encourages employees to work together across different departments and functions, providing them with tools and resources to facilitate collaboration and communication.

Overall, Oracle's organizational culture is characterized by a commitment to innovation, customer focus, excellence, collaboration, and teamwork. These values are reflected in the company's products, services, and operations, and are embraced by its employees at all levels of the organization.

## 1.2 Project Presentation

### 1.2.1 Objectives

Our team is developing a constrained mode assistant tailored for specific tasks. This assistant is designed to be reliable in production, capable of handling complex, multi-step tasks. The steps

in this process are executed sequentially according to a loosely directed graph, which allows users to backtrack if necessary. These steps can perform various functions such as gathering user-specific context, soliciting user input, generating text, and more, utilizing LLMs, tools, or hardcoded rules. Transitions between steps are clearly defined to enhance reliability. An essential early step in this process is the "choice selection step," which enables users to choose the subsequent step based on their specific task requirements.

Given our use of LLMs, a question that naturally follows is:

**Q1: How can we evaluate and improve prompts for the choice selection step?**

These assistants are not limited to conversation and can help with all kinds of tasks, and one that is particularly handy for programmers is code generation.

However, using AI-generated code, trained on diverse internet sources, in proprietary software could introduce legal risks related to copyright and licensing. It is crucial to ensure that our systems are legally compliant and adhere to licensing norms. This concern leads us to the following question:

**Q2: How can we efficiently verify if an AI-generated code snippet originates from a specific, curated knowledge codebase<sup>1</sup>?**

### 1.2.2 Proposed solutions

The first challenge involves establishing a reliable way to evaluate the performance of templates used in various assistant steps. We need this to enable easy comparison of different prompts by running evaluations on various instances of the steps, then we need to optimize the prompt so that the LLM accurately classifies user tasks based on a comprehensive, verified dataset. We want to achieve:

- **High Accuracy** The LLM should consistently classify text into the correct category class given a set of categories to choose from.
- **Low Latency:** The model should generate responses quickly to ensure a smooth user experience.

In the context of ensuring legal compliance in the use of AI-generated code, the second challenge focuses on the ability to accurately identify and verify code within large codebases or across various software projects that align with specific legal and functional requirements. The significance of this research is rooted in several challenges out of which we will tackle:

- **Code Search System Creation:** Development of a highly scalable and predictive code search system.

---

<sup>1</sup>Knowledge codebase: A database that includes comprehensive collections of code, documentation, and related meta-data. This database is constructed from various sources, often including vast amounts of free open-source software.

- **Scalability:** the ability of the code search retrieval system to handle growing amounts of work or its potential to be enlarged to accommodate that growth. In practical terms, this involves how well a system can manage large volumes of data or queries efficiently.
- **Semantical Alignment:** Ensuring that the code retrieved matches the intended meaning or purpose, rather than just syntactical correctness. It involves understanding the functional goal of a code query and ensuring the results align with these goals in terms of what the code is meant to accomplish.

## 1.3 Project Coordination

### 1.3.1 Scrum Methodology

Our team uses a straightforward and flexible framework known as Scrum to enhance collaboration in software development projects. Scrum helps teams handle complex projects by dividing them into smaller, manageable tasks, completed in short cycles called sprints. This method focuses on teamwork, accountability, and steady progress toward a clear goal.

Scrum involves three key roles: the Product Owner, the Scrum Master, and the Development Team. The Product Owner ensures the product provides maximum value and manages the list of tasks (Product Backlog). The Scrum Master supports the team in following Scrum practices, removing obstacles, and ensuring the team is effective. The Development Team, a self-organizing group, is responsible for completing the tasks and delivering product increments.

Scrum operates through fixed-length cycles called sprints, usually lasting two to four weeks. At the beginning of each sprint, team members choose tasks from the Product Backlog to complete during the sprint. The sprint concludes with a Sprint Review, where completed work is presented to stakeholders, and a Sprint Retrospective, where the team evaluates their process and identifies areas for improvement.

To keep track of progress during a sprint, the team uses a Sprint Board, a visual tool with columns representing stages of task completion such as "To Do," "In Progress," and "Done." Each task is represented by a card that moves across the columns as it progresses. The Sprint Board provides a clear overview of the team's work, highlighting any bottlenecks or issues and ensuring transparency. This helps keep everyone aligned on what needs to be done and what has been completed.

### 1.3.2 Operational Tools

#### Slack

Slack is a messaging app designed primarily for business communication, offering a structured way to manage communications within a company and catering to various organizational needs.

Public channels in Slack are accessible to all workspace members and are used for general communications, including company-wide announcements and casual interactions. These channels foster an open environment where coworkers can share information freely and engage in light-hearted conversations, contributing to a positive workplace culture. I enjoy participating in these channels as they help me stay connected with colleagues and stay informed about what's happening across the company.

For more personalized communication, direct messages allow individuals to have private conversations with peers, mentors, or managers. This is especially important for discussing private matters or receiving individual guidance. I often use direct messages for quick check-ins with my team or to seek advice from my mentors.

Private channels are particularly useful for keeping project notes and reminders, accessible only to the team members involved. I personally rely on private channels to track notes, links, and updates during meetings, so that I can easily access all the relevant information I need.

## **Jira**

I have found Jira to be an indispensable tool in our development process. Originally designed for tracking bugs and tickets, Jira has evolved into a comprehensive work management platform suitable for a wide range of use cases, from managing requirements and test cases to supporting Agile development methodologies.

In our projects, we leverage boards, which serve as our central task management hubs. These boards map tasks to customizable workflows, making sure that everyone on the team has clear visibility into the status and progress of each task, and keeping our projects on track.

One of the features I particularly appreciate is Jira's time tracking capabilities. This allows us to monitor the amount of time spent on each task, helping us identify areas for improvement and make adjustments as needed.

## **Bitbucket**

I rely heavily on Bitbucket, a version control repository hosting service by Atlassian, which supports both Mercurial and Git systems. Bitbucket integrates seamlessly with Jira, another Atlassian product, enhancing our project management and development workflows.

This integration is particularly valuable as it allows for automatic updates to Jira issues based on code changes in Bitbucket, such as commits, pull requests, and branch creations. This means that as soon as I push a commit or open a pull request, the corresponding Jira issue is automatically updated. This feature enables me and my team to track project progress directly within Jira, linking code changes to specific tasks or issues. This not only simplifies tracking but also improves project transparency.

By linking our code changes with specific Jira tasks, we can easily see the status and progress of our work, ensuring that everyone on the team is on the same page. This streamlined approach to project management and version control helps us maintain a high level of organization and efficiency in our development process.

## **Confluence**

I find Confluence, another tool from Atlassian, essential for facilitating team collaboration through document creation and sharing. Confluence is particularly useful for maintaining project documentation, tracking decisions, and collaborating on project specifications.

One of the standout features of Confluence is its integration with Jira and Bitbucket. This integration allows us to link documents directly to specific Jira tasks and Bitbucket repositories. For instance, when we create project specifications or decision logs in Confluence, we can link them to the corresponding Jira issues and Bitbucket branches, which that all project-related information is easily accessible and up-to-date.

Confluence keeps all relevant information in one place and directly connected to its implementation in the code which supports our efforts to deliver software efficiently and collaboratively.

## **Conclusion**

In this chapter, we provided a comprehensive overview of Oracle Corporation, along with a broad look at the project objectives, proposed solutions, and coordination approach. We also outlined the operational tools used. Next, we will examine the project context, discussing objectives, competitive analysis, and requirements, to provide a thorough understanding of the project's technical and strategic dimensions.

# Evaluating and Improving Prompts for

## Choice Selection

don't mention choice selection in the title, but rather smth more

### Introduction

In this chapter, we outline the project's core problem, its context, and the proposed solution. We also define the project's objectives, competitive analysis, requirements, and methodology, followed by an experimental setup and results analysis.

## 2.1 Project Context

Despite the potential of AI and ML technologies to revolutionize enterprise app development, their impact has not yet resulted in substantial productivity gains. Currently, AI implementations are more focused on personal assistants, such as chatbots, rather than team-oriented assistants that could enhance collaborative efforts within enterprises. Additionally, these technologies struggle to manage the complexity of enterprise app data and logic, particularly in understanding roles and permissions, which are crucial for enterprise applications. Furthermore, AI and ML have not yet enabled users to perform the customizations they desire or provided the deep analytical insights they seek, limiting their effectiveness in meeting enterprise needs.

### 2.1.1 General Objective

Based on the analysis of two primary types of agent frameworks:

- **DAG agents:** These are highly directed and structured, but they are cumbersome to configure and often underwhelming in practical applications.
- **Open agents:** These are impressive in their capabilities, but they can be unpredictable and lack robustness for production environments.

Our aim is to integrate the strengths of both frameworks into our package. By potentially leveraging existing technologies, we seek to bridge the gap between these two types of agent frameworks, achieving a balanced and effective solution.

### 2.1.2 Competitive Analysis of Langchain

Langchain offers easy-to-use, open-source agents and two ways to define sequences of actions:

- **Chains:** Actions are hardcoded in code, such as processing user input through a Retrieval-Augmented Generation (RAG) system, a prompt template, a Language Model (LLM), and producing an output.
- **Agents:** LLMs determine the next actions, processing input through a prompt template, an LLM, a RAG system, a calculator, and then producing the output.

Langchain agents offer several key features, including maintaining chat history across sessions and defining tools for LLMs, such as search tools and code execution. The complexity of tools depends on the model's capabilities, with more powerful models like GPT-4 able to use multi-input tools. Agents can invoke multiple tools in a sequence or in parallel during a single user interaction, allowing for open-ended, dynamic responses based on LLM output. Each agent consists of an LLM, a set of tools for executing actions, and a prompt template that stores previous tool invocations and outputs, as well as chat history. An output parser interprets the LLM's output to determine when to invoke a tool or complete the task.

However, a significant drawback of Langchain is the difficulty in swapping underlying LLMs due to the heavy dependence of tools and prompts on specific LLMs, often necessitating the creation of new chains for different models. Fixed actions in chains limit misuse and ensure a specified order but reduce flexibility and require updates when new tools are introduced. Conversely, dynamic actions in agents provide flexibility but can be more easily manipulated by malicious users. Implementing safeguards and tightly controlling the tools available to the LLM can help mitigate these risks.

### 2.1.3 Assistant Capabilities

The user would interact with the assistant through a multi-round chat-like interface. Each round, the user states their request, and the model can either:

- **Propose some action:** This can involve asking for confirmation from the user or not.
- **Execute some action in the background.**
- **Answer with some text:**
  - To ask for more information.
  - To provide a summary of the actions it just made.
  - To provide a text answer to a question.

### 2.1.4 Main Components of the Assistant

We will specifically focus on the **constrained mode** from now on and we would like to explore the set of abstractions we need to support that mode:

- **Assistant steps** define what to do when in a particular state. These steps can use LLMs to perform required actions, tools, or potentially just hardcoded rules. They are what get executed by the assistant graph.
- **Steps and transitions between steps** are strictly defined for more reliability. Extensions for less strictly defined orders are left for the future.
- **Assistant graph** defines what steps the assistant can take and what transitions from one step to another are allowed.
- **Tools** provide access to external data or functionalities (e.g., a calculator tool). Tools can be used by any step.
- **Conversation** corresponds to the execution of the assistant graph one or more times and the interactions with the user during those executions.

### 2.1.5 Pre-defined Steps for the Assistant

We provide here a small list of steps to give a general idea of how they work:

- **User Input Step:** The model outputs a message (potentially a question) and the user has to provide some answer.
- **Output Step:** A text is created from the template and the text input and put inside the messages for the user to see.
- **Prompt Execution Step:** it generates an output using a LLM based on a template, the user input and the user history.
- **Choice Selection Step:** This is the subject of interest; it determines which next step to run. It takes a list of potential steps with descriptions at construction time and user input at invocation time then the LLM is prompted to give which next task to execute based on the user context.

### 2.1.6 Addressing Operational Challenges

#### 2.1.6.1 Security

Users might attempt to perform prompt injections to gain full control, which is particularly relevant for the creative step. They might also try to execute code on the machine that hosts the assistant when the step can use a code execution tool. APIs that can be used in the code tools need to be sandboxed and vetted to ensure security.

### 2.1.6.2 Reliability

LLM answers might not always match the expectations of the user or developer. Specifically, the creative step might take actions (such as calling tools) that could have detrimental effects on the user. To mitigate this, we aim to offer reversibility of model actions so that users can revert to a previous good state.

### 2.1.6.3 Safety

Additionally, the LLM might misunderstand the user's intent and execute harmful functions that were not planned by the user. It is important to note that LLMs are currently not very secure and are susceptible to various attacks, including code execution injections and backdoors.

## 2.2 Benchmark Design

### 2.2.1 Problem Statement

At present, conducting a straightforward comparison between two templates on a single step is cumbersome, as it requires the creation of:

- Two steps
- Two graphs
- Two assistants
- $2 \times N$  graph scripts, where  $N$  is the number of different user inputs to be tested

Providing APIs to automate and simplify this process would encourage more frequent benchmarking and facilitate the improvement of existing templates.

Priority	User	Story
High	Assistant Developer	I want to be able to compare two templates of my step in a few lines of code using a dataset of expected inputs and expected outputs of an assistant.

Table 2.1: Benchmark Design User Story

### 2.2.2 Competitive Analysis

In the context of benchmarking assistants, PromptFlow and DialogFlow offer extensive prompt comparison toolboxes that streamline the evaluation process with built-in functionalities and APIs. This allows developers to easily compare and optimize different templates and configurations.

Conversely, LangChain requires users to perform prompt comparisons manually, necessitating additional effort to set up and execute benchmarks. This manual approach can lead to inconsistencies and increased time investment.

### 2.2.3 Proposed Solution

A testing framework for agents has been designed to ensure they can follow scripted conversations. However, there is a need for a reliable method to benchmark the performance of templates used in various assistant steps. These templates are currently optimized based on perceived performance, but a systematic approach is required to evaluate and enhance them in terms of both performance and efficiency.

The proposed solution introduces a new class, *PromptBenchmarker*, which facilitates benchmarking of specific steps to improve template performance. This class allows for easy comparison of different prompts by running benchmarks on incomplete lists of steps, which are filled with various instances of the steps to be compared. It also uses a dataframe containing inputs and parts of the expected output to compute a basic score.

The *PromptBenchmarker* class is built on top of the script runner and has been demonstrated to effectively compare and improve existing prompts, such as the *ChoiceSelectionStep* template.

### 2.2.4 Prompt Benchmarker

We propose the creation of a new *PromptBenchmarker* class, which utilizes the graph script runner with several key differences:

- Instead of taking a list of scripts, it accepts a dataframe with the following columns:
  - Name of the script
  - ID of the script interaction
  - User input of the interaction
  - The string expected to be found in the assistant’s response. This can be extended to functions or checks for more advanced metrics in the future.
- Instead of taking a list of assistants, it will take a graph or list of steps, including one placeholder. This placeholder will be filled by the *PromptBenchmarker* with steps instantiated using the provided arguments.

The *PromptBenchmarker* class will handle the creation of all graphs, instantiation of all assistants, creation of graph scripts, and execution of these scripts to finally return the reports for the run.

## 2.3 Methodology

As discussed previously, our objective is to develop an effective prompt for the LLM that ensures accurate classification of given text inputs. To achieve this, we will follow a structured methodology leveraging the proposed benchmark design.

### 2.3.1 Ground Truth Dataset

We will curate a comprehensive ground truth dataset that includes various text inputs and their corresponding categories or domains. This dataset will serve as the benchmark against which the LLM’s responses will be evaluated.

#### 2.3.1.1 Dataset Overview

The dataset we will use is known as multi-domain question rewriting (MQR) dataset[7], it is constructed from human contributed Stack Exchange question edit histories. The dataset contains 427,719 question pairs which come from 303 domains.

Stack Exchange is a question answering platform where users post and answer questions as a community. If questions do not meet their quality standards, members of the community often volunteer to edit the questions. Such edits typically correct spelling and grammatical errors while making the question more explicit and easier to understand.

The questions in MQR are mostly English sentences. Having questions from 303 Stack Exchange sites makes the MQR dataset cover a broad range of domains.

The dataset used *PostHistory.xml* and *Posts.xml* tables of each Stack Exchange site data dump. If a question appears in both files, it means the question was modified. The most up-to-date Stack Exchange questions are treated as a well-formed questions and its older version from *PostHistory.xml* as ill-formed. The latter only keeps one edit for each question, so the MQR dataset does not contain duplicated questions.

The ill-formed versions are not limited to questions but can also be queries, commands or statements. They often contain grammatical, spelling or stylistic errors.

Ill-formed	Well-formed	Category
Spaghetti carbonara, mixing	How to mix a spaghetti carbonara?	cooking
Ethical Investing... where to begin?	How to begin ethical investing?	money
charging canon sx 700 battery through powerbank	Can I charge a Canon SX 700 battery using a mobile powerbank?	photo
H1B Visa consulate interview timeline	What is the timeline for an H1B visa consulate interview?	expatriates
Hanging weight from drywall ceiling	How much weight can I hang from a drywall ceiling?	diy

Table 2.2: Examples of pairs of ill-formed and well-formed questions from the MQR dataset.

#### 2.3.1.2 Data Preprocessing

- **Rectifying category names:** We made replacements to the way category names are written. Here are a few examples: ‘codereview’ is replaced with ‘Code Review’, ‘softwareengineering’: ‘Software Engineering’, ‘cstheory’: ‘Computer Science Theory’, ‘ai’: ‘Artificial Intelligence’, ‘datascience’: ‘Data Science’

The reason why this change is important is because when the tokenizer deals with OOV (out of vocabulary) tokens, it could run the risk of producing semantically insignificant embeddings for the model which will hinder its performance. Here is an example of tokens generated by the Mixtral tokenizer for a word and its rectified version:

- Tokens for *'Computer Science'*: [*'\_Computer'*, *'\_Science'*]
- Tokens for *'computerscience'*: [*'\_computers'*, *'ci'*, *'ence'*]

This was just to demonstrate that the two words that are seemingly similar, are treated differently by the the embedding model by proxy.

- **Picking relevant categories:** We manually pick categories such that they relate to a technical fields for what we envision from what the user wants help with when using the assistant. We ended up with 20 categories: *'Physics'*, *'Math'*, *'Web Apps'*, *'Quantum Computing'*, *'Electronics'*, *'Academia'*, *'Cryptography'*, *'Computer Science'*, *'Linguistics'*, *'Biology'*, *'Security'*, *'Android'*, *'Graphic Design'*, *'Arduino'*, *'Artificial Intelligence'*, *'DevOps'*, *'Computer Graphics'*, *'Data Science'*, *'Bioinformatics'*, *'Internet of Things'*, *'Literature'*.
- **Attaching id's to categories:** Under the lens of minimizing the number of tokens generated by the model, we plan to attach id's to categories and have the model guess the id instead, so the process starts by creating a mapping of unique category names to numerical identifiers from a given dataset.
- **Generating multiple datasets:** Subsequently, multiple datasets are generated from a primary pool of data, which is a random subset of the large MQR dataset. Each generated dataset ensures an equal representation of each category by randomly sampling the specified number of entries from each.  
The goal here is to see how well a prompt template does with increasing complexity (number of categories).
- **Data Curation:** We manually verify a significant amount of samples if the text corresponds to the right category to minimize mistakes and resolve some of the ambiguity.

### 2.3.2 Prompt Format

Prompt engineering is not just about designing and developing prompts. It's an important skill to interface, build with, and understand capabilities of LLMs. You can use it to build new capabilities like augmenting LLMs with domain knowledge and external tools.

We can achieve a lot with simple prompts, but the quality of results depends on how much information we provide it and how well-crafted the prompt is. A prompt can contain information like the instruction or question we are passing to the model and include other details such as context, inputs, or examples. We can use these elements to instruct the model more effectively to improve the quality of results. A standard prompt has the following format: *<Question>?* or *<Instruction>*. We can format

this into a question answering (QA) format, which is standard in a lot of QA datasets, as follows: Q: *<Question>*? A:

When prompting like the above, it's also referred to as **zero-shot prompting**, i.e., we are directly prompting the model for a response without any examples or demonstrations about the task we want it to achieve. Some large language models have the ability to perform zero-shot prompting but it depends on the complexity and knowledge of the task at hand and the tasks the model was trained to perform good on. With some of the more recent models we can skip the "Q:" part as it is implied and understood by the model as a question answering task based on how the sequence is composed.

A prompt contains any of the following elements:

- **Instruction** - a specific task or instruction you want the model to perform
- **Context** - external information or additional context that can steer the model to better responses
- **Input Data** - the input or question that we are interested to find a response for
- **Output Indicator** - the type or format of the output.

To demonstrate the prompt elements better, here is a simple prompt that aims to perform a text classification task:

```
Classify the text into neutral, negative, or positive
Text: I think the food was okay.
Sentiment:
```

In the prompt example above, the instruction correspond to the classification task, *Classify the text into neutral, negative, or positive*. The input data corresponds to the *I think the food was okay.* part, and the output indicator used is *Sentiment:*. Note that this basic example doesn't use context but this can also be provided as part of the prompt. For instance, the context for this text classification prompt can be additional examples provided as part of the prompt to help the model better understand the task and steer the type of outputs that you expect.

### 2.3.3 Benchmarking prompts for choice selection

To develop effective prompts for the LLM, we will focus on the *ChoiceSelectionStep*. This step is crucial as it determines the next action based on user input.

We will utilize the *PromptBenchmarker* class to facilitate the benchmarking process. Our evaluation will prioritize the following metrics to ensure the LLM's performance is robust:

- **Accuracy:** The correctness of the LLM's classifications against the ground truth dataset.
- **Latency:** The time taken by the LLM to generate responses.

## 2.4 Experimental Setup

- **Model:** We use **Mixtral-8x7B-Instruct-v0.1**, a model that was released together with the base Mixtral 8x7B model. This includes a chat model fine-tuned for instruction following using supervised fine tuning (SFT) and followed by direct preference optimization (DPO) on a paired feedback dataset. Let's fix some model parameters:

- **max\_new\_tokens:** to manage the number of tokens the model generates, prevent long or irrelevant responses and control costs. We set it to 2 tokens, because for this particular tokenizer, each digit corresponds to one token, since we did not exceed two-digits for categories (20 of them), it's enough to have 2 tokens as a limit.
- **Other model configurations:** We leave the default parameters as they are.

```
"top_k": 50,  
"top_p": 0.95,  
"temperature": 0.7,  
"repetition_penalty": 1.03,
```

We do not want to make the model more deterministic because we want to have an idea of how confident it is from its answers i.e how consistently it generates the same output for a given prompt.

- **Inferencing Engine:** We use **vLLM**, a high-throughput and memory-efficient inference and serving engine for LLMs, it integrates nicely with HuggingFace models, it provides continuous batching of incoming requests, and has optimized CUDA kernels, etc.
- **Datasets:** We will test on the datasets generated from the MQR dataset. We generated 6 randomly sampled datasets with 20 samples per category across all of them. It starts with 5-categories-set, it finishes at 20-categories-set with an increasing step of 3. We will refer from now to each dataset as **k\_catg\_set.csv**, with *k* being the number of categories contained in the dataset.
- **Prompt templates:** We will start with a baseline prompt template and we will iteratively refine it while exploring new techniques, adapting them, and keeping the working structures of the prompt. We will refer to different prompts with their names, e.g the baseline prompt template will be referred to as *BASELINE\_PT*.

Element	Description
Instruction	You are a helpful assistant. Your goal is to understand what category the user input belongs to the best.
Context	The available categories are loaded using a Jinja loop.
Input Data	human_question: question from the human that could be categorized as category 1
Output Indicator	The format you need to follow is: "" user_input: question from the human that could be classified as category that has an attached id of 1 category_id: 1 "" Begin! Remember to only answer with the category id.

Table 2.3: Elements of *BASELINE\_PT*

## 2.5 Experiments and Results

### 2.5.1 Sensitivity to user input

By using ill-formed questions as user input to test the LLM, we simulate a more realistic scenario where users may not always provide well-structured, grammatically correct, or accurately spelled queries. This approach allows us to evaluate the model's robustness and ability to handle arbitrary input, which is closer to real-world user interactions where errors and imperfections are common.

We run the *BASELINE\_PT* 5 times on the **8\_catg\_set** dataset and we take the average of running 5 times. We measure the accuracy which reflects simply how many examples were correctly predicted by the model.

User Input	Accuracy
well-formed	74%
ill-formed	69%

Table 2.4: Accuracy according to different quality of user input

This is not surprising given that the text classification task is not one that requires coherence; intuitively, specifying a few relevant words regardless of the structure of the sentence should point the model to what category the input belongs to the best. That's good news, because in production if the user makes mistakes, the model accuracy will decrease by little to none.

### 2.5.2 Using a short prompt template

We manually verbalize 5 prompts of the same structure e.g in this case 5 short prompts. The best performing one is kept to be tested against the other prompt templates with different structures.

The same approach is used for the coming experiments and we will only be showing the best performing one that was kept for the sake of brevity.

Element	Description
Instruction	You are a classification expert. Categorize the given input into one of these predefined choices:
Context	The available categories are loaded using a Jinja loop.
Input Data	<i>This is left empty</i>
Output Indicator	<i>This is left empty</i>

Table 2.5: Elements of *SHORT\_PT*

We run the *SHORT\_PT* on the `8_catg_set.csv` dataset, this is how it looks like when it's after invocation time, followed by the results from the run:

<p><b>LLM prompt:</b> You are a classification expert. Categorize the given input into one of these predefined choices:</p> <p>2: Android                      4: Artificial Intelligence      7: Computer Graphics  8: Computer Science    10: Data Science                      11: DevOps  12: Electronics                      21: Web Apps</p> <p><b>Input:</b> How to calculate growth function for a threshold function like this?  category_id:</p> <p><b>LLM answer:</b> 4</p>
---

Category	Accuracy	Latency (ms)
Computer Graphics	0.80	0.05
Android	0.73	0.06
Artificial Intelligence	0.60	0.05
Data Science	0.60	0.05
DevOps	0.60	0.05
Electronics	0.60	0.05
Computer Science	0.53	0.06
Web Apps	0.53	0.05
Mean	0.62	0.05

Table 2.6: Summary of *SHORT\_PT* on `8_catg_set`

We would typically want a prompt length that is small because that lowers costs of computation and execution time but this metric is considered more in the later stages of tuning.

As this is tested on a well formed user input, *SHORT\_PT* suffers somewhat noticeably compared to the *BASELINE\_PT* with 74% (averaging 5 runs is quite significant to confirm that).

### 2.5.3 Using Chain-of-Thought prompt template

Chain-of-thought (CoT)[45] prompting enables reasoning capabilities in LLMs through intermediate reasoning steps. We can combine it with one-shot prompting to get better results tasks that require reasoning before responding.

The same as what was mentioned before, we look at the best performing rephrased prompt template that uses Chain-of-Thought, which is essentially explaining in steps how the output is produced given some input.

Element	Description
Instruction	You are a classification expert. Provided a question, your goal is to understand it and categorize it within a predefined set of choices, from which you cannot deviate.
Context	The available categories are loaded using a Jinja loop. <code>###</code> Let's start with an explanation: <code>&gt;&gt;&gt;</code> Inquiry: Loss jumps abruptly when I decay the learning rate with Adam optimizer in PyTorch <code>&lt;&lt;&lt;</code> the id number 4 corresponds to the Artificial Intelligence category, which is the correct category that the inquiry belongs to. So your answer will be <code>category_id: 4###</code>
Input Data	<i>Inquiry: Loss jumps abruptly when I decay the learning rate with Adam optimizer in PyTorch</i>
Output Indicator	<i>category_id: 4</i> (where 4 corresponds to the Artificial Intelligence category).
Delimiters	<code>###</code> wraps the complete prompt scenario including the inquiry and the correct category assignment. <code>&gt;&gt;&gt;</code> and <code>&lt;&lt;&lt;</code> signs enclose the specific user inquiry input and the corresponding category id output to be processed or demonstrated within the template.

Table 2.7: Elements of *COT1SHOT\_PT*

The delimiters part was directly inspired from the Mistral documentation<sup>1</sup> about prompting capabilities, the idea is that delimiters specify the boundary between different sections of the text. In our example, we used `###` to indicate examples and `<<<>>>` to indicate customer inquiry.

We run the *COT1SHOT\_PT* on the `8_catg_set.csv` dataset, here is how it looks after invocation time, followed by the results from the run:

<sup>1</sup>[https://docs.mistral.ai/guides/prompting\\_capabilities/classification](https://docs.mistral.ai/guides/prompting_capabilities/classification)

**LLM prompt:** You are a classification expert. Provided a question, your goal is to understand it and categorize it within a predefined set of choices, from which you cannot deviate. The available categories are:

2: Android                      4: Artificial Intelligence      7: Computer Graphics  
 8: Computer Science      10: Data Science              11: DevOps  
 12: Electronics              21: Web Apps

###

**Inquiry:** Loss jumps abruptly when I decay the learning rate with Adam optimizer in PyTorch the id number 4 corresponds to the Artificial Intelligence category, which is the correct category that the inquiry belongs to.

So your answer will be category\_id: 4

###

You will only respond with the category id.

<<< **Inquiry:** Fiding websites that have been 'Liked' on facebook? >>>

**Answer:** category\_id: 21

Category	Accuracy	Latency (ms)
Electronics	0.93	0.06
Android	0.87	0.06
Web Apps	0.87	0.05
Computer Graphics	0.80	0.06
DevOps	0.73	0.06
Artificial Intelligence	0.67	0.05
Computer Science	0.67	0.06
Data Science	0.53	0.06
Mean	0.76	0.06

Table 2.8: Summary of *COT1SHOT\_PT* on *8\_catg\_set*

Let's look at the performance of all prompt templates across categories to see if we can draw some conclusions.

### 2.5.4 Performance according to different prompt templates

These results are tested on the *8\_catg\_set.csv*

Categories	Accuracy			Duration		
	<i>BASELINE_PT</i>	<i>SHORT_PT</i>	<i>COT1SHOT_PT</i>	<i>BASELINE_PT</i>	<i>SHORT_PT</i>	<i>COT1SHOT_PT</i>
Android	1.00	0.73	0.87	0.05	0.05	0.05
Electronics	0.99	0.60	0.93	0.05	0.05	0.05
DevOps	0.99	0.60	0.73	0.05	0.05	0.05
Computer Graphics	0.80	0.67	0.80	0.05	0.05	0.05
Computer Science	0.53	0.36	0.67	0.05	0.05	0.06
Web Apps	0.45	0.53	0.87	0.05	0.05	0.05
Data Science	0.66	0.60	0.53	0.05	0.05	0.06
Artificial Intelligence	0.40	0.60	0.67	0.05	0.05	0.05
<b>Mean</b>	0.74	0.62	<b>0.76</b>	0.05	0.05	0.05

Table 2.9: Performance Metrics Across Different Categories

The first remark is that the average time taken to execute one prompt remains the same, that's due the max new tokens parameter we set to 2. More tokens to generate would mean more execution time.

The second remark is that prompts work well if given more information as seen in the difference in accuracy between the short template and the other templates. We will see that the outperforms both on the average by a reasonable margin. Part of the reason why we chose the `8_catg_set.csv` is to avoid any bias and overly specific judgement.

The third and most important remark is that the model's accuracy from one category to another, a behavior consistent in all prompt templates, could decrease by a large margin, that's due to the fact that the model is confused about which is the correct category to pick, meaning closeness of categories causes model ambiguity. An example of that would be that, for instance, *Artificial Intelligence* and *Computer Science* are close subjects, and that would allow questions that are general enough to fit in both to be hard to classify.

### 2.5.5 Summary

Now we summarize the results of all prompts templates on all the datasets in one three-dimensional graphic containing a set of points, each point is the mean of 5 runs, and represents the accuracy of each prompt template with numbers of categories, we are particularly interested in how well they perform under such increasing complexity:

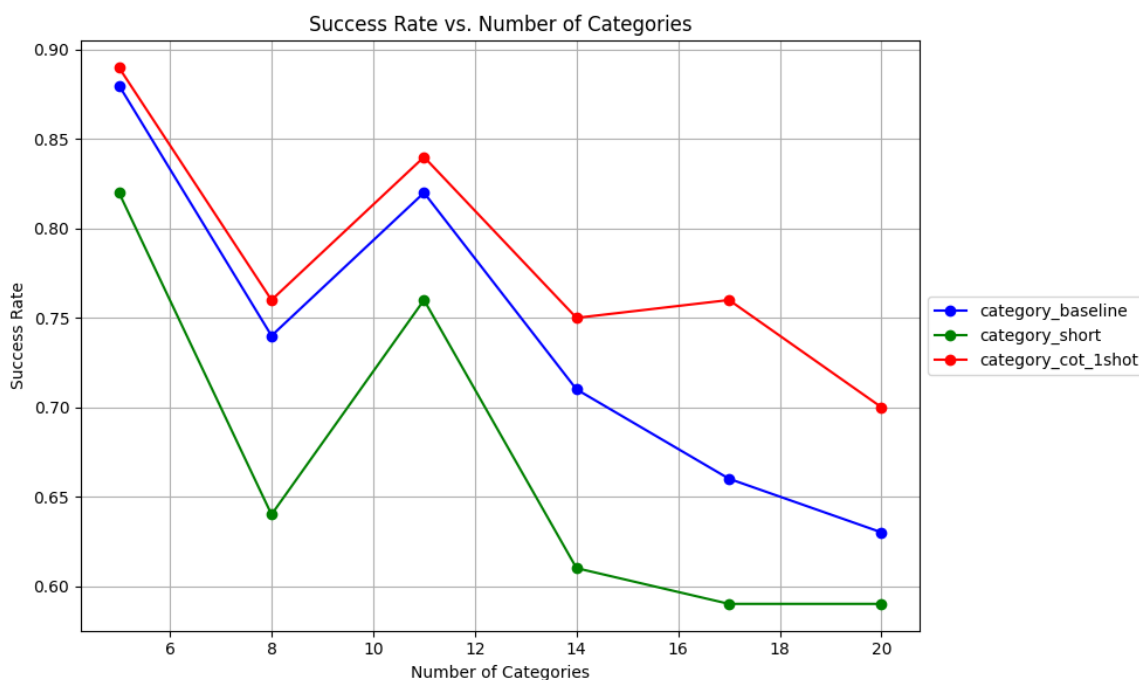


Figure 2.1: Prompt accuracy with increasing complexity

The first noticeable behavior is that CoT prompting outperforms exceptionally when the number of categories is highest.

The second observation is that the dip in the set with 8 categories emphasizes the point that's been made about the ambiguity problem, this implies that the random samples that were picked are arbitrarily hard to classify, due to the nature of how they were collected.

It's important to note that, in production, we could possibly use a model that is stronger and could have a potentially high accuracy like GPT-4.

To some extent, this experiment is more or less a testament that prompt engineering works.

## 2.5.6 Future Work

Even though the text classification task remains a relatively easy task for an LLM, there are definitely areas for improvement:

- **Prompt Versioning** which helps in managing different iterations and choosing the most effective one.
- **Automatic Verbalizer** can reduce manual efforts in choosing the right words for the prompts.
- **Better Data Curation**, obtaining clean data with varied categories requires a lot of effort and manual work, but it plays a big role in our evaluation of the model's ability to classify text accurately.

## Conclusion

This chapter concludes our exploration of integrating different agent frameworks to enhance AI and ML applications in enterprise environments. We have looked at the benchmark design and later focused on prompt engineering to improve precision particularly. Moving forward, we will address code search, a challenge closely linked to scalable code clone detection. This issue becomes particularly relevant as users increasingly rely on LLM-based assistants for generating code, which may lead to potential licensing infringements.

# Code Search for License Compliance

## Introduction

This chapter addresses the challenges inherent in the code search problem, introduces a cutting-edge methodology to tackle these challenges, and reviews state-of-the-art reference results in this domain.

## 3.1 Project Presentation

### 3.1.1 Proposal

Our primary goal is to assess the feasibility of developing a system that can identify similarities between specific code snippets and a large repository of such snippets. The section is organized as follows:

- **Problem Statements**, where we define the issue we aim to resolve and outline the associated challenges.
- **Competitive Landscape**: We briefly review Black Duck, a key competitor known for helping organizations manage code security and compliance risks effectively.
- **Proposed Solution**: We present our methodology, grounded in a review of academic literature that discusses state-of-the-art solutions.

### 3.1.2 Problem Statements

The pervasive issue of code similarity is a focal point in software engineering research, with applications such as code clone detection playing a crucial role. Code clone detection involves identifying replicated code fragments within a software system, which not only helps in managing code quality but also in mitigating licensing conflicts. Instances where software licensing violations stem from code cloning are well-documented, illustrating the complications that arise when code is copied from software under stringent licenses to those with more lenient terms. This can lead to significant

legal challenges, such as the well-publicized case where Linksys (a Cisco company) reused code from the Linux kernel and BusyBox in their proprietary WRT54G router, infringing the GPL license. Typically, software companies mitigate this risk by conducting code clone detection and license analysis services provided by software auditing firms like Black Duck Software.

The integration of generative AI tools in software development has introduced new complexities. Tools like OpenAI's ChatGPT and GitHub Copilot generate code snippets that may inadvertently include code with restrictive licenses, posing potential legal issues. This underscores the need for robust mechanisms to automatically identify and manage licensing risks inherent in code generated by AI, ensuring compliance with intellectual property laws. The integration of such systems into continuous integration pipelines is essential to maintain legal and ethical standards in software development.

The presence of several large-scale online code databases, such as GitHub along with other repository and versioning systems which contain potentially billions of lines of code, poses significant challenges to code similarity detection. This necessitates the development of scalable tools capable of efficiently identifying code clones across extensive databases. Traditional methods, such as tree-based or text-based detection tools, struggle with scalability and often cannot manage the vast quantities of code found in modern databases.

The primary challenge of measuring similarity of source code on the Internet-scale is the quadratic execution time from  $n - to - n$  pairwise comparisons of code fragments or the expensive tree or graph comparisons in traditional code similarity detection approaches. Thus, we are seeking a scalable approach that reduces or avoids such comparisons while still accurately locate similar code fragments.

Addressing the code search problem is essential for Oracle to ensure that all AI-generated code within its products and systems is legally compliant and adheres to licensing requirements. This commitment to software integrity and compliance safeguards Oracle's products against potential legal risks.

The code search problem is defined as the challenge of identifying suitable code within large codebases or across various software projects that meet specific requirements or queries. The significance of this research is rooted in two primary challenges:

- **Scalability of Search Systems:** Due to the expansive size of codebases, manual searches become impractical, posing significant challenges to the scalability of retrieval systems.
- **Semantical Alignment:** It is essential to retrieve code snippets that are not only syntactically similar but also semantically aligned with the intended purposes of the code.

### 3.1.3 Competitive Landscape

Code assistants are AI-driven tools designed to enhance developer productivity and ensure code consistency, especially in key Oracle technologies like Java, SQL, and Oracle Cloud Infrastructure (OCI). These assistants use fine-tuned Large Language Models (LLMs) based on open-source models trained on various licensed codebases. The generated code might resemble open-source code, leading to potential licensing issues. To address this, we will conduct a competitive analysis on tools like Blackduck to identify licensing problems in code snippets and mitigate these risks.

Black Duck, a software solution developed by Synopsys, empowers organizations to effectively manage security, compliance, and operational risks associated with the use of open-source software. This tool scans codebases to identify open-source components and their versions, assesses potential security vulnerabilities, and evaluates license compliance and code quality. Black Duck methodically indexes global open-source projects with its comprehensive knowledge codebase, ensuring accurate component identification within applications.

The platform aggregates data from over 7 million projects across more than 30,000 repositories and forges, sourcing from public repositories such as GitHub and SourceForge, language-specific repositories like RubyGems and PyPI, and individual project sites such as MySQL. It helps safeguard intellectual property and mitigates risks of license noncompliance by tracking over 2750 open-source licenses.

Further insights into Black Duck's detection capabilities reveal nuanced behaviors under various code modifications:

- The presence of a function definition is not a prerequisite for Black Duck to identify code matches. Nonetheless, diminishing the number of initial code lines progressively impairs its detection efficiency.
- Sensitivity to Commentary Density: Black Duck successfully identifies matches with the inclusion of few code comments. However, the detection capability deteriorates when that escalates.
- A critical threshold exists where beyond a certain extent of code modification or obfuscation, Black Duck fails to trace the snippet back to its source.

### 3.1.4 Proposed Solution

The goal is to study code cloning in large-scale source code data and develop a scalable clone search approach to address challenges from such cloning. To achieve the goal, the following objectives are set.

To develop a service for detecting code cloning to and from online sources such as GitHub, we need a tool with scalability. Based on the literature, index-based techniques offer high scalability [3], but still suffer from low detection accuracy on clones with challenging modifications such as added,

deleted, or relocated statements (type-3 clones) [44].

Based on this, we offer a hybrid solution that attempts to solve both problems:

- An **index-based code clone search system** that leverages OCI search capabilities for high scalability and primarily targeting text similarity (identical and lexical changes).
- An **embedding-based code clone search pipeline** that has a retrieval phase using a pretrained model for code and a classification phase using a strong LLM, that targets functionally-aware code snippets (syntactical and semantical changes).

## 3.2 Literature Review of Code Clone Detection

### 3.2.1 Program Similarity

Computer program similarity can be assessed at three distinct levels, as demonstrated by Zhang et al. [50]. At the **purpose level**, two programs might share the same function, such as sorting numbers in ascending order. If they utilize the same sorting algorithm, they are considered similar at the **algorithm level**. Finally, similarity at the **implementation level** occurs when two programs use the same method to execute the algorithm.

However, determining whether two programs share the same purpose or algorithm is often challenging or even impossible, as noted by Koschke [23]. Consequently, most software similarity assessments are conducted at the implementation level. This can involve various forms of the software, including source code, compiled code, syntax trees, or metrics extracted from the software. We will focus on software similarity based on source code. Thus, when we discuss "code similarity," we primarily refer to similarity at the implementation level as represented by the software's source code.

Code similarity is a concept that permeates several research fields and goes by various names, including code clones, software plagiarism, copy-and-paste code, similar code, code duplication, and software redundancy. Despite the different terminology, the underlying concept remains consistent.

### 3.2.2 Source Code similarity

Source code similarity is a measure used to evaluate the degree of resemblance between two code snippets based on their text, syntax, and semantics. Although a universal definition of code similarity does not exist, it is commonly quantified by a real number  $s$ , where a higher score indicates greater similarity between two code fragments,  $c_1$  and  $c_2$ . Yamamoto et al. [48] describe software system similarity as the proportion of similar element pairs relative to the total element pairs, using a specific similarity metric. Qinqin and Chunhai [27] suggest that this definition also applies to the similarity of program source codes.

It's crucial to distinguish between measuring code similarity and identifying similar or cloned codes. The former is a broad concept, while the latter is an application of measuring code similarity, where highly similar code instances are flagged as clones based on a threshold. In fact, clone detection stands out as the most notable application, extensively covered in literature and tools. Advances in clone detection methodologies can further enhance other applications of code similarity measurement.

### 3.2.2.1 Code Clones Definition

Code clones refer to similar fragments of code and represent a prominent application of code similarity within software engineering, constituting an active area of research. Code cloning involves developers duplicating source code, either as-is or with modifications, and is a routine practice in this emerging AI-assisted software development.

There are various interpretations of what constitutes code clones. Kamiya et al. [21] describe code clones as two segments of code that are either "identical" or "similar." In a more precise definition, Baxter et al. [1998] categorize clones as two program fragments that are exactly alike, and refer to slightly different fragments as "near-miss clones. Jiang et al. [18] offer another perspective, identifying a clone pair as "two code fragments that exhibit a similar tree representation up to a certain degree of similarity."

### 3.2.2.2 Code Similarity

The term "similarity" used in many clone definitions varies widely, as it is defined independently by different researchers, leaving it subject to interpretation by each specific clone detection method. The measurement unit, whether it be a code fragment, snippet, or portion, is also not consistently defined. The way code fragments are treated varies from one tool or technique to another, depending on what best suits their algorithms.

### 3.2.2.3 Granularity of Code Snippets

The granularity of code snippet pairs in this definition can vary across different programming abstractions, such as statements, blocks, methods, classes, packages, or components within a program. Additionally, the measurement of code similarity may be confined to a single project or extend across multiple projects, utilizing cross-project similarity measurement techniques. Practically, the outcomes of code similarity measurements are often presented as a **vector**, listing the ranked similarities between a given code snippet and all other snippets at the same entity level.

### 3.2.2.4 Code Clone Terminology

We adopt widely recognized terminology concerning code clones as referenced in several seminal works [36], [39], [5], [36]. Below is an explanation of the terminology used:

- **Code Fragment:** A segment of code identified by a tuple consisting of the source file, and the starting and ending line numbers.
- **Clone Pair:** Two code fragments that are associated with a specific type of similarity, i.e., Type-1, Type-2, Type-3, or Type-4.
- **Type-1 clones:** Code fragments that are exactly identical except for differences in whitespace, layout, and comments.
- **Type-2 clones:** Code fragments that are syntactically identical but may differ in identifiers, literals, types, and formatting.
- **Type-3 clones:** Similar code fragments that have modifications such as added, removed, or relocated statements.
- **Type-4 clones:** Code fragments that may not be syntactically similar but perform the same function or produce the same output, thus sharing semantical similarity.

This terminology forms the basis for discussing and analyzing code clones throughout this work.

### 3.2.3 Code similarity detection methods

A considerable portion of research on code similarity focuses on developing methods for identifying similar code segments and their implications for software development. Since the 1970s, a plethora of tools and techniques for measuring code similarity have emerged. Comprehensive reviews of these tools and techniques are available in works by [36], [23], [34], and [33]. These tools use various approaches to detect similar code fragments within or across programs. Drawing parallels to a survey on clone detection techniques by [36], this work categorizes the discussed code similarity approaches and tools into groups based on the abstraction level of the source code, such as metric-based, text-based, token-based, tree-based, and other techniques.

#### 3.2.3.1 Metric-based methods

Metric-based methods represent source code as vectors of various code metrics, using the vector space model theory [40]. The similarity between two pieces of code is then measured by comparing their vectors. The accuracy of this method depends greatly on the quantity and quality of the defined metrics for the programming language. Computing these metrics for large programs is reasonably time-efficient [29], [19]. The following is a table that lists common source code metrics used for code similarity measurement and clone detection, showing that researchers typically use a limited set of metrics.

In the initial stages of code similarity measurement, metric-based approaches such as software metrics were used to compute software similarity. However, due to their superficial analysis and lack of deep structural understanding of programs, these methods have largely been surpassed by more advanced techniques.

Metric
Lines of code
Number of variables declared
Total number of operators
Number of loops (for, while, and do-while)
Cyclomatic complexity
Number of function calls
Number of conditional statements
Number of return statements

Table 3.1: Common metrics used for clone detection

### 3.2.3.2 Text-based methods

The simplest and earliest technique for measuring code similarity involves treating the source code as a text document [12]. Text-based methods view the source code as a sequence of strings [30], comparing two code snippets to identify the longest common sequence of strings, which are then reported as clone instances. This comparison is usually done on raw code without preprocessing, although in some cases, white spaces and comments are removed before matching sequences. The main advantage of text-based methods is their efficiency and applicability to any programming language without modification.

These methods particularly excel at identifying clones that result from direct copying and pasting without modification (Type-1 clones) but struggle to detect clones with syntactic or semantical alterations (Type-2, Type-3, or Type-4 clones). To improve their detection capabilities for syntactic and semantic similarities, additional techniques are incorporated into the similarity computation process. Below, we discuss the methods and tools that utilize textual comparisons:

**Longest Common Subsequence (LCS) algorithm** rooted in edit distance theory, identifies similar segments between two strings based on the number of insertions and deletions required. Originally developed to compare amino acid sequences, LCS has been adapted for various string comparisons, including source code.

Several code similarity detection tools implement LCS. NiCad [8] is a clone detector that compares lines of source code. This tool uses the TXL (Turing eXtender Language) grammar to parse and transform specific code segments accurately. NiCad’s process involves multiple stages such as pretty-printing, abstraction, normalization, and filtering of code. In the similarity computation phase, NiCad employs LCS to compare pre-processed lines of code from two programs.

Although most text-based tools are good at detecting cloned or plagiarized code fragments that are identical or nearly identical with only minor modifications, they are less effective at identifying similar code with added, deleted, or relocated statements [9]. According to a study by Roy et al. [36], while most text-based tools can fully or partially detect Type-1 clones, they generally fail to find clones of Type-2, Type-3, and Type-4, with the exception of NiCad.

### 3.2.3.3 Token-based methods

In token-based methods, source code is converted into sequences of tokens, which are then compared to find common subsequences [28], [47]. These techniques are more robust to code changes, such as modifications in identifier names, than text-based methods. Token-based approaches are efficient enough for frequent use during the software development process [38]. However, they do require additional processing time for token recognition and replacement. These methods have shown strong performance in identifying type I and type II code clones, as well as detecting type III clones with many edits, known as large-gap clones [31].

Token-based approaches represent a higher level of abstraction from the source code by converting programs into tokens. These tokens, often normalized to remove textual discrepancies, serve as abstract representations to capture only the structural elements of the code. This methodology is favored for its simplicity, flexibility, and scalability in code matching. Below are some of the ideas that utilize token-based representations.

**Normalized Tokens** allow for the overlooking of changes in formatting and identifiers, enabling cross-language clone detection. This is achieved by normalizing code written in various programming languages to a uniform set of abstracted tokens. Tools like Plague and Sherlock utilize this approach for structuring program profiles and conducting incremental comparisons across textual and token-based analyses.

**N-grams** are contiguous sequences of  $n$  items from a given data set, are extensively used in computational linguistics and have been adopted for partial matching in text and code analysis. For example, MOSS uses n-grams along with a document fingerprinting algorithm to detect software plagiarism by converting source code strings into n-grams, computing a hash sequence, and creating fingerprints for comparison.

### 3.2.3.4 Tree-based methods

In tree-based methods, source code is converted into a parse tree or abstract syntax tree (AST) using a programming language parser [37], [35], [15], [49]. Similar code is identified by searching for similar subtrees within the parse tree or AST [26]. This approach is robust to code changes like different block structures and parentheses, but generating parse trees or ASTs for large codebases is time-consuming and requires a specific parser for each programming language [24]. Additionally, matching subtrees is computationally expensive, often reaching an upper complexity bound of  $O(N^3)$  [4], prompting the integration of various optimization techniques to reduce this complexity.

Notable tools like CloneDR utilize ASTs combined with hashing techniques to efficiently locate clones by reducing the comparison domain to manageable buckets. This method allows for the detection of near-miss clones through specialized hash functions that overlook identifiers. Similarly, Deckard [18] employs characteristic vectors to represent ASTs and uses Locality Sensitivity Hashing

(LSH) to cluster similar trees, significantly reducing computational demands. Falke et al. [13] introduced an innovative approach using suffix trees transformed from syntax trees to perform clone detection in linear time, proving significantly faster than traditional AST-based methods while only detecting Type-1 and Type-2 clones.

### 3.2.3.5 Learning-based methods

We opted to discuss learning-based methods as a separate category due to the increasing prevalence of machine learning in recent years [17]. Machine learning techniques, summarized in this section, have shown promising results in code similarity measurement and clone detection tasks [46], [25], [51], [16]. However, they require large, clean datasets of code clones to function correctly, which are not available for all programming languages. Most approaches rely on existing code clone detection tools to prepare the necessary data, which can be unreliable due to inherent errors in these tools.

Learning-based approaches aim to classify or cluster similar code fragments by training on datasets of known similar and dissimilar codes [22], [25]. Cesare et al. [6] employed conventional classification models to identify clones at the package level, reporting Random Forest as the best-performing model. Joshi et al. [20] utilized the DBSCAN clustering algorithm to identify Type I and II clones at the function level. Keivanloo et al. [22] used the k-means clustering algorithm to distinguish true and false Type III clones across various dissimilarity thresholds, a common issue in non-learning methods. They evaluated clustering quality using the Friedman method for different values of the  $k$  parameter. However, their approach cannot detect Type IV clones.

### 3.2.3.6 Hybrid methods

Hybrid methods combine two or more base approaches to mitigate the limitations of individual methods. It is common that text-based and token-based methods are frequently combined with graph-based and tree-based methods. While text-based and token-based methods are efficient but less effective, graph-based and tree-based methods are effective but less efficient. Their combination thus provides a balance of efficiency and effectiveness for code similarity measurement. Additionally, combining token-based and text-based methods enhances robustness while maintaining the efficiency of text-based methods in detecting types I, II, and III clones [42], [1], [41].

## 3.2.4 Embeddings for Code Clone Detection

Embeddings are low-dimensional spaces where high-dimensional vectors are projected. These embeddings encapsulate the rich semantic details from the inputs, positioning related inputs closer in this feature space based on their similarity.

This section explores popular word embedding algorithms and introduces code embeddings. We aim to use code embeddings for code clone detection and retrieval, reducing reliance on computa-

tionally intensive architectures.

### 3.2.4.1 Word Embeddings

represent each word in a text. Tomas Mikolov’s introduction of the Word2Vec algorithm in 2013 popularized this concept, followed by the GloVe (Global Vectors for Word Representation) algorithm by Pennington et al. in 2014, which enhances word-to-word relationship modeling using a co-occurrence matrix. These representations facilitate various applications like machine translation, text summarization, and text classification.

Common to all word embedding algorithms is the proximity of similar words and the distance of unrelated words in the embedding space, as illustrated in Fig. 2.3. This spatial arrangement is achieved through specific loss functions in the training process, such as the negative sampling loss used by Word2Vec.

However, both Word2Vec and GloVe face limitations with polysemy—the multiple meanings of a word. For instance, the word "bank" can imply different entities depending on the context, which these algorithms fail to distinguish by assigning a singular representation. This limitation led to the development of contextual embeddings in recent models like BERT and ELMo, which provide context-specific word embeddings.

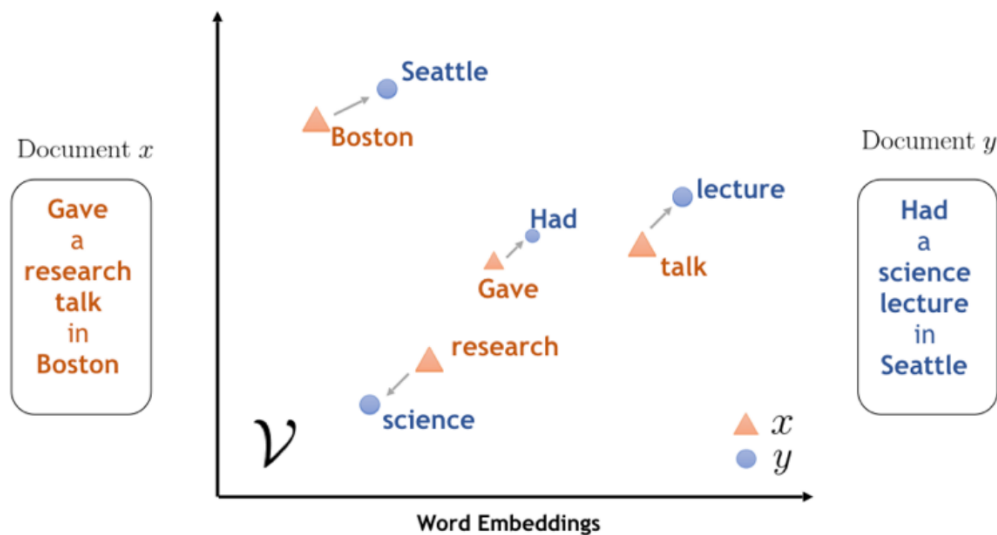


Figure 3.1: Word2Vec embeddings for documents X, Y

### 3.2.4.2 Code Embeddings

We plan to generate code embeddings using the CodeT5 architecture for each piece of code in our repository. These embeddings will be instrumental in clone detection and retrieval, aiming to lessen the dependency on heavy computational architecture during inference. For clone detection, we will assess the similarity between code pair embeddings. For retrieval, we will use nearest neighbor search techniques to match query code embeddings with those pre-computed for the dataset.

### 3.2.4.3 Embedding-based methods for code clone detection

Embedding-based methods leverage neural networks (NN) [46],[2], [32] to tackle clone detection and retrieval challenges. These approaches vary in how they represent programs and in training different NN architectures. While deep learning models excel in precision and recall for clone detection, they struggle with scalability in retrieval tasks. This section summarizes existing studies that utilize deep learning for these purposes, also highlighting a key survey paper that discusses the advantages and disadvantages of these methods [43], .

**CodeBERT** CodeBERT [14] is an advanced neural network tailored for code understanding and generation. It harnesses neural representations of natural and programming languages, using the BERT [10] architecture as its base. It is pre-trained with two objectives: Masked Language Modeling and Replaced Token Detection. This training involves processing both the program and its description to yield a unified, information-rich representation. The model is then fine-tuned for various applications such as code search, clone detection, and code summarization.

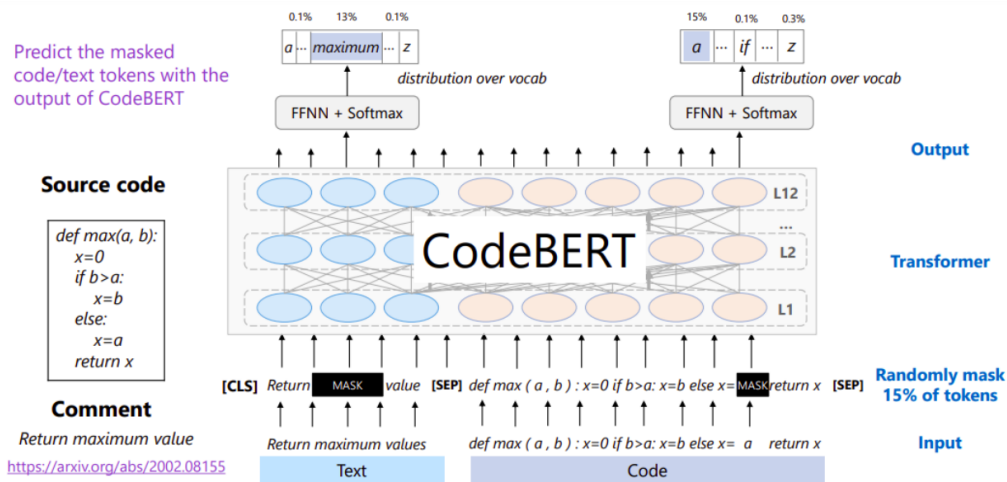


Figure 3.2: CodeBERT training using the Masked Language Modeling objective

**GraphCodeBERT** To address CodeBERT's limitation in capturing structural code information, GraphCodeBERT [15] was developed. Unlike CodeBERT, which represents code as a simple sequence of tokens, GraphCodeBERT incorporates the code's structural information via the program's data flow graph. This model encodes semantic information about variable dependencies and is pre-trained on three tasks: Masked Language Modeling, predicting dependency edges in code structures, and aligning representations between the source code and its structure. GraphCodeBERT enhances CodeBERT's performance and achieves top results in four downstream tasks.

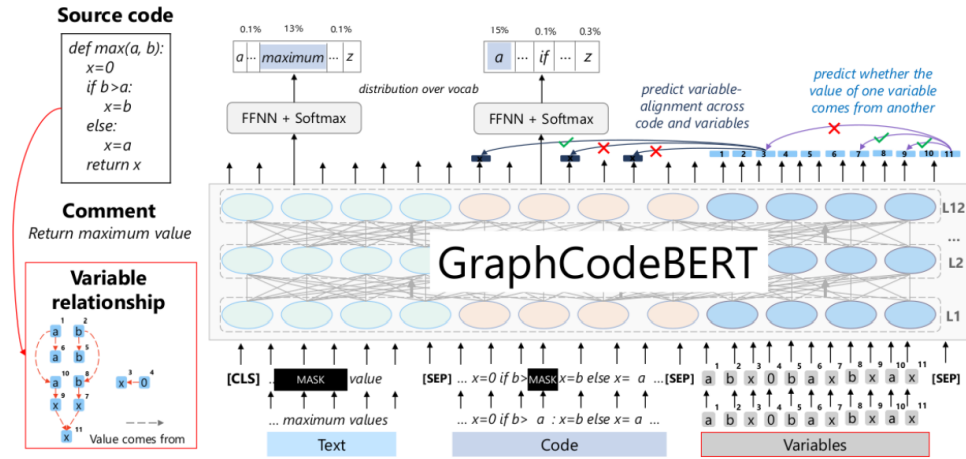


Figure 3.3: GraphCodeBERT training using the Masked Language Modeling objective

**CodeT5** Wang et al. introduced CodeT5 [70], a Transformer-based encoder-decoder model that focuses on semantic nuances captured through user-defined identifiers. Extending the T5 model [49]—originally for natural language tasks—CodeT5 is adapted for code-related applications. It introduces an identifier-aware pre-training objective to handle critical token-type information from the programming language. CodeT5 excels in fourteen code-related sub-tasks within CodeXGLUE [41], including code clone detection, and significantly surpasses previous architectures.

CodeT5 adapts the T5 architecture by integrating code-specific knowledge, enhancing its understanding of code. It processes code alongside its comments as sequential inputs. As depicted in the figure, CodeT5 is pre-trained through a series of objectives.

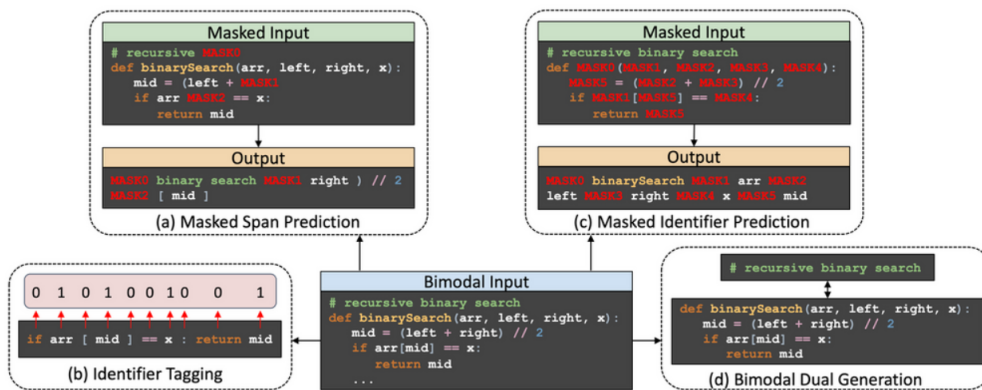


Figure 3.4: Pretraining tasks for CodeT5

### 3.3 Methodology

#### 3.3.1 Code similarity detection phases

The code clone detection process encompasses several phases:

1. **Preprocessing:** This involves the removal of irrelevant code segments to isolate disjoint fragments of source code units, for example, removing whitespaces and comments.
2. **Transformation:** The resulting data undergoes various transformations. For text-based approaches, we only keep the initial form since we're strictly doing full-text search. The source code is then converted into tokens, parsed into an abstract syntax tree (AST), transformed into a program dependency graph (PDG), metrics are computed, and identifiers are normalized.
3. **Matching:** detecting clones by comparing the transformed units from the previous phase. Methods such as suffix trees, suffix arrays, pattern matching, hash values, and metric evaluations are used.
4. **Formatting:** Clones identified in the matching phase are mapped back to their original locations within the source code based on file locations and line numbers.

Since the different methods we saw have different representations for code, they use different tokenization techniques, on this table we can see the type of transformation and the state that we converted the code to:

Technique	Transformation	Intermediate State
Text-based	Removes comments and whitespace	Normalized code
Token-based	Tokenization	Sequence of tokens/lines
Tree-based	Parsed to tree	AST nodes
Graph-based	Parsed to graph	PDG nodes
Metric-based	Parsed to AST or PDG	Scalar metric values

Table 3.2: Transformations and Intermediate States

### 3.3.2 Code Similarity Evaluation

The evaluation of code similarity search is based on three main factors: **efficacy**, **efficiency**, and **scalability**.

- **Efficacy** is assessed through metrics such as precision, recall, F1 score, and accuracy:
  - **Precision** measures the accuracy of the detected similarities, focusing on minimizing false positives.
  - **Recall** quantifies the ability of the technique to identify all relevant instances of code similarities.
  - **F1 Score** provides a balance between precision and recall by calculating their harmonic mean.
  - **Accuracy** evaluates the overall correctness of the technique across all samples.
- **Efficiency** refers to the duration a search system takes to complete its search for clones in a specific subject system, constrained by the available computational resources.

- **Scalability and Execution Time** assesses the ability of a search system to analyze large subject systems using a limited amount of computational resources, while it does not exceed resource capacity or require excessively long execution times.

### 3.3.2.1 Efficacy

Let us consider the universe,  $U$ , representing all possible clone pairs (every pair of code fragments) within the system. The set of all true clone pairs,  $T$ , is identified by a clone oracle, which hypothetically judges if a pair of code fragments is a clone. The complement of  $T$  in  $U$ ,  $F = U - T$ , represents the false clone pairs—those not identified as clones by the oracle. On the other hand,  $D$  denotes the set of clone pairs detected by the clone detector in the system.

From the perspective of clone detection, this delineation divides the universe into four categories:

- **True Positives:** True clones that are successfully detected. (Desirable for recall improvement)
- **False Positives:** False clone pairs incorrectly identified as true. (Reduces precision)
- **True Negatives:** False clone pairs correctly not reported. (Enhances precision)
- **False Negatives:** True clone pairs that are missed by the tool. (Reduces recall)

Recall is calculated as the proportion of true clone pairs detected, represented by the intersection of  $T$  and  $D$ :

$$\text{Recall} = \frac{|D \cap T|}{|T|} = \frac{\text{true positives}}{\text{true positives} \cup \text{false negatives}} \quad (3.1)$$

Precision is the proportion of detected clone pairs that are true, given by the intersection of  $D$  and  $T$ :

$$\text{Precision} = \frac{|D \cap T|}{|D|} = \frac{\text{true positives}}{\text{true positives} \cup \text{false positives}} \quad (3.2)$$

To enhance recall, the tool should maximize true positives and minimize false negatives. Conversely, improving precision requires maximizing true positives while minimizing false positives.

### 3.3.2.2 Search Efficiency

Instead of using clone detectors, another approach is to use search heuristics that are different from the clone detectors. Ideally, these heuristics should have high recall but may have lower precision, and they help identify potential clones. These identified clones are then checked manually before they are added to the reference corpus. This method reduces the amount of manual searching needed, making it more efficient. However, it's possible that some true clones might be missed and not added to the corpus.

### 3.3.2.3 Search Scalability and Execution Time

The performance of a clone detector, including how long it takes to run (execution time) and how well it works with large software systems (scalability), is very important. Execution time is simply how long the detector needs to finish analyzing a software system. Scalability refers to the ability of the clone detector to handle large systems without running out of memory or disk space, and without

taking too much time. Scalability also means that the time it takes to run the detector can increase as the system gets bigger or more complex.

Users don't like to wait long for analysis results, so it's crucial that the execution time is quick. This is especially important when clone detection is part of a larger tool-chain, like during the build process of software. The clone detector must be fast enough so it doesn't hold up the build, especially when software updates are frequent.

These detectors are not only used within single software systems but also to find clones across different software systems, branches, and even across an organization's whole collection of software. There are new uses emerging for clone detection, like analyzing very large repositories that contain millions or billions of lines of code. It's crucial to see how both current and new clone detectors manage these large scales.

The effectiveness of a clone detector in terms of execution time and scalability depends a lot on the software system it's analyzing and the type of computers it runs on. We're particularly interested in how our method will perform on high-power computers like servers or clusters.

### 3.3.3 Targeting Textual Similarity Types

Since we're targeting Java, the method parsing is accomplished using a Java parser, while tokenization is handled by the Antlr4 lexer using Java 8 grammar.

The methodology we could aim for is leveraging Oracle OpenSearch which is a high-performance, open-source, distributed full-text search engine designed for scalable code indexing and retrieval.

Feature	Benefit
Built-in Search Capabilities	Offers a number of features to help you customize your search experience such as Full-text querying, Autocomplete, Scroll Search, customizable scoring and ranking, and more.
SQL Query Syntax	Provides the familiar SQL query syntax. Use aggregations, group by, and where clauses to investigate your data.
Search Support in SQL	Enables you to use the familiar SQL query syntax while getting access to the rich set of search capabilities such as fuzzy matching, boosting, phrase matching, and more.
k-NN search	Using Machine Learning, run the nearest neighbor search algorithm on billions of documents across thousands of dimensions with the same ease as running any regular OpenSearch query.

Table 3.3: Search-Related Features in OpenSearch

The features of OpenSearch are designed to support diverse and demanding environments, re-

flecting a blend of established technologies and innovative approaches. The built-in search capabilities enhance user experience by supporting intricate query types and intelligent search functionalities, such as autocomplete, which predicts user input to streamline searches, and scroll search, which enables efficient browsing through large datasets.

Moreover, the SQL Query Syntax feature in OpenSearch allows users to perform complex data analysis using familiar SQL commands, which simplifies the transition for new users from traditional databases to OpenSearch.

Here is a table of some of the full-text search queries we plan to use:

Query Type	Description
match	The default full-text query, which can be used for fuzzy matching and phrase or proximity searches.
match_phrase	Similar to the match query but matches a whole phrase up to a configurable slop.
simple_query_string	A simpler, less strict version of query_string query.

Table 3.4: Selected Query Types

In OpenSearch, the abstraction that encompasses text analysis is referred to as an **analyzer**. Each analyzer consists of the following components, applied sequentially:

- **Character filters:** A character filter receives the original text as a stream of characters and can add, remove, or modify characters. For example, a character filter might strip HTML markup from text, transforming `<p><b>Actions</b> speak louder than <em>words</em></p>` into “Actions speak louder than words.”
- **Tokenizer:** This component takes the stream of characters processed by the character filter and splits the text into individual tokens (usually words). For example, it might split text based on whitespace, resulting in the tokens [Actions, speak, louder, than, words].
- **Token filters:** After tokenization, the token filter processes the stream of tokens. It can modify tokens (e.g., by converting all characters to lowercase), remove tokens (e.g., stopwords like “than”), or add tokens (e.g., adding synonyms such as “talk” for “speak”).

Opensearch uses the Apache Lucene search library, which provides highly efficient data structures and algorithms for ingesting, indexing, searching, and aggregating data. Apache Lucene’s core advantage lies in its ability to perform high-performance full-text search. Its sophisticated document indexing process allows for fast retrieval of textual data, making it an ideal choice for extensive databases and applications requiring complex search functionalities.

Lucene achieves this through the use of inverted indexing, a technique where each document is broken down into its constituent terms and a reverse index is created to list documents containing

those terms. This makes the search process incredibly efficient, as searching for a term involves looking up the index rather than scanning through all documents.

Moreover, Lucene provides advanced search features such as ranked searching, which prioritizes results based on relevance to the query. This is complemented by powerful query types including boolean, phrase, wildcard, and proximity queries, allowing for precise and flexible search capabilities. Lucene also supports multi-language and multi-field searching, enhancing the robustness of search applications across diverse datasets and use cases.

It also incorporates customizable scoring and ranking algorithms. Users can modify these algorithms based on their specific needs, enabling tailored search experiences that align with organizational objectives or user preferences. Additionally, Lucene's API supports real-time indexing, which allows for the immediate addition of new documents to the index without significant performance loss. This is crucial for dynamic environments where information is constantly updated.

### 3.3.4 Targetting semantical similarity types

To tackle the complexities of code search, we propose the establishment of a code search pipeline system, underlining a two phase methodology:

- **Phase 1:** Retrieve the most relevant and similar code snippets by calculating a similarity score based on their embeddings.
- **Phase 2:** Deploy advanced pre-trained Large Language Models (LLMs) that are capable of interpreting code to classify the candidate code snippets as either strongly similar or not.

This approach ensures efficiency and precision in discerning code relevance and similarity within our system.

#### 3.3.4.1 Code Search for Retrieval

We can utilize advanced models such as CodeBERT or CodeT5+ which are effective at understanding the syntax, structure, and semantics of code. To manage these code representations, we can implement an efficient storage system that allows us to quickly identify highly similar code snippets. The system's ability to scale and handle large volumes of data efficiently is critical. In this phase, achieving high recall is more important than precision, meaning we aim to capture as many potentially relevant code snippets as possible, even if some are not perfectly matched.

#### 3.3.4.2 Classification for Code Clone Matching

We plan to deploy sophisticated pre-trained language models like GPT-4, LLaMA, Vicuna, Falcon, or StarChat-beta to classify a subset of code identified as similar during the retrieval phase. The goal is to determine if any of the snippets are significantly similar, turning this task into a binary classification where each snippet is closely compared with others that are structurally similar. This process includes analyzing the structure, syntax, and semantic intent of the code, recognizing this as

a method for detecting cloned code. Given the high computational demands, this detailed analysis is not feasible for the entire database, underscoring the importance of the initial retrieval phase. The success of the entire project hinges on the quality of outcomes from the retrieval phase. We strive to maintain a balance between high precision and high recall to ensure both the accuracy and effectiveness of the matching process. The following is the whole pipeline illustrated in a figure:

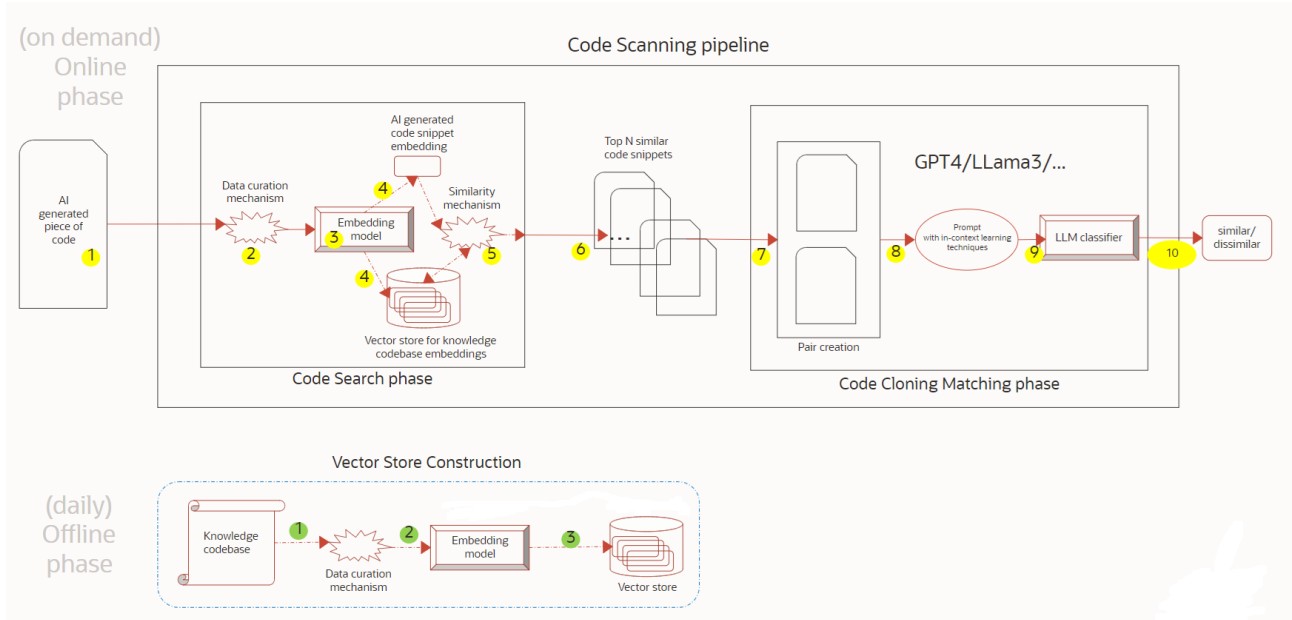


Figure 3.5: Two-phase Methodology

Here is a detailed walkthrough:

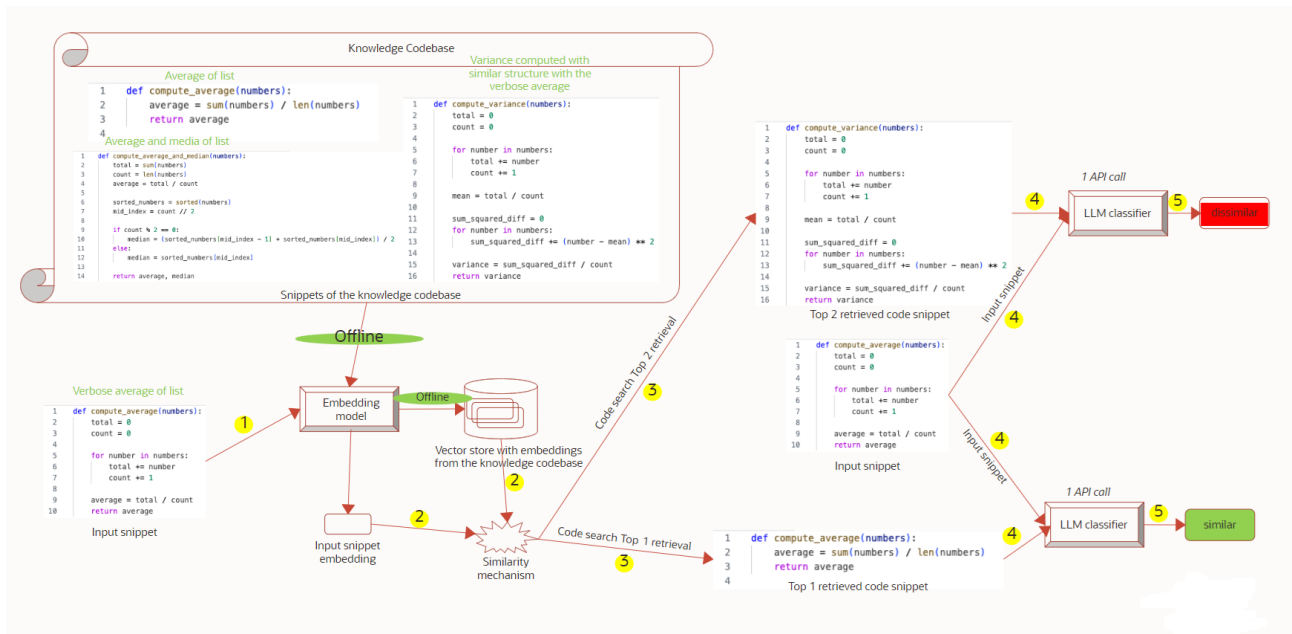


Figure 3.6: Detailed Example Walkthrough

### 3.3.5 Reference State-of-the-Art Results

CodeBERT, a bimodal pre-trained model for programming language (PL) and natural language (NL), excels at code retrieval tasks. It learns general-purpose representations supporting various NL-PL applications like natural language code search and code documentation generation. Studies show that CodeBERT performs exceptionally well in detecting Type-1 and Type-4 code clones, achieving high recall rates. Additionally, fine-tuning CodeBERT can significantly enhance its performance on unseen functionalities, improving recall rates by a significant margin on different datasets

MODEL	RUBY	JAVASCRIPT	GO	PYTHON	JAVA	PHP	MA-AVG
NBow	0.4285	0.4607	0.6409	0.5809	0.5140	0.4835	0.5181
CNN	0.2450	0.3523	0.6274	0.5708	0.5270	0.5294	0.4753
BiRNN	0.0835	0.1530	0.4524	0.3213	0.2865	0.2512	0.2580
SELFATT	0.3651	0.4506	0.6809	0.6922	0.5866	0.6011	0.5628
RoBERTA	0.6245	0.6060	0.8204	0.8087	0.6659	0.6576	0.6972
PT w/ CODE ONLY (INIT=S)	0.5712	0.5557	0.7929	0.7855	0.6567	0.6172	0.6632
PT w/ CODE ONLY (INIT=R)	0.6612	0.6402	0.8191	0.8438	0.7213	0.6706	0.7260
CODEBERT (MLM, INIT=S)	0.5695	0.6029	0.8304	0.8261	0.7142	0.6556	0.6998
CODEBERT (MLM, INIT=R)	0.6898	0.6997	0.8383	0.8647	0.7476	0.6893	0.7549
CODEBERT (RTD, INIT=R)	0.6414	0.6512	0.8285	0.8263	0.7150	0.6774	0.7233
CODEBERT (MLM+RTD, INIT=R)	<b>0.6926</b>	<b>0.7059</b>	<b>0.8400</b>	<b>0.8685</b>	<b>0.7484</b>	<b>0.7062</b>	<b>0.7603</b>

Figure 3.7: Results on natural language code retrieval. [14]

Using open-source LLMs for clone detection yields superior results in identifying Type-3 and Type-4 clone pairs when relying solely on a simple prompt. However, it does exhibit slightly poorer performance when detecting Type-1 and Type-2 clone pairs compared to existing tools. Notably, GPT-3.5-Turbo and GPT-4 stand out with the highest recall and accuracy rates across nearly all clone types. Also, the clone detection performance of GPT-3.5-Turbo and GPT-4 can be improved by requiring models to provide clone type, similarity, reasoning, and similarity lines. Using one-step chain-of-thought prompts allows the models to analyze code pairs and intermediate reasoning, leading to better clone detection.

Methods	T1	T2	VST3	ST3	MT3	T4	Precision
GPT-3.5-Turbo	1	0.98	0.98	0.94	0.87	0.36	0.77
GPT-4	0.99	1	0.98	0.98	0.92	0.25	0.89

Table 3.5: Recall and Precision on Clone Type Reasoning [11]

Methods	Language	T1	T2	VST3	ST3	MT3	T4	Precision
GPT-3.5-Turbo	Java	1	0.57	0.85	0.78	0.59	0.09	0.95
GPT-3.5-Turbo	Python	0.99	0.94	0.61	0.46	0.41	0.22	0.99
GPT-3.5-Turbo	C++	0.99	0.99	0.68	0.44	0.33	0.16	1
GPT-4	Java	1	0.98	0.99	0.94	0.77	0.15	0.96
GPT-4	Python	1	0.99	0.99	0.99	0.90	0.72	1
GPT-4	C++	1	1	0.97	0.95	0.87	0.67	1

Table 3.6: Methods Recall and Precision [11]

## Conclusion

In conclusion, this chapter addressed the challenges of searching for code by introducing advanced methods and reviewing the latest tools in this area. We explored the importance of detecting similarities in code. Various tools were assessed, and a combined approach was proposed to improve both the speed and accuracy of code searches.

# General Conclusion

This thesis has delved into the potential of advanced agent frameworks to enhance artificial intelligence and machine learning applications, focusing on enterprise environments. We examined two primary areas: the capabilities of AI assistants in handling complex tasks and the challenges associated with code similarity detection in software development.

Firstly, we explored the development and optimization of AI assistants. By analyzing current technologies, we identified significant gaps in the ability of these systems to manage complex data and user interactions effectively. Our research highlighted the necessity for more dynamic and adaptable agent frameworks that can handle the nuanced demands of enterprise applications. The proposed solutions aimed to integrate the precision of directed agents with the flexibility of open agents, creating a robust system that improves user interaction and task execution.

In the realm of software development, we addressed the critical challenges of code generation and similarity detection. The importance of developing scalable tools for managing extensive codebases was emphasized, particularly in ensuring compliance with software licensing standards—a vital aspect given the rise of generative AI tools like ChatGPT and GitHub Copilot. Our proposed benchmark design for frequent, automated testing of AI assistants not only enhances their functionality but also contributes significantly to software engineering by refining evaluation methodologies.

Furthermore, we tackled the code search problem, crucial for maintaining legal compliance and software integrity in development environments. The integration of scalable search systems and semantic alignment techniques ensures that developers can efficiently locate and manage code, thereby mitigating potential legal and security risks.

In conclusion, this thesis has laid a foundational framework for future advancements in AI and ML applications within enterprise settings. The methodologies and solutions presented aim to bridge current technological gaps, providing a pathway for future research and development. As we continue to refine these systems, the potential to significantly enhance both the functionality of AI assistants and the efficiency of software development practices is vast and inspiring.

# Bibliography

- [1] A. Agrawal and S. K. Yadav. A hybrid-token and textual based approach to find similar code segments. *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, 7 2013.
- [2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: learning distributed representations of code. *Proceedings of the ACM on programming languages*, 3(POPL):1–29, 1 2019.
- [3] F. Author and S. Another. Index-based code clone detection: incremental, distributed, scalable. In *IEEE Conference on Software Engineering*, pages 1–10. IEEE, September 2010.
- [4] I. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 1998.
- [5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE transactions on software engineering*, 33(9):577–591, 9 2007.
- [6] S. Cesare, Y. Xiang, and J. Zhang. *Clonewise – Detecting Package-Level Clones using machine learning*. 1 2013.
- [7] Z. Chu, M. Chen, J. Chen, M. Wang, K. Gimpel, M. Faruqui, and X. Si. How to ask better questions? a large-scale multi-domain dataset for rewriting ill-formed questions, 2019.
- [8] J. R. Cordy and C. K. Roy. The NiCad Clone Detector. *2011 IEEE 19th International Conference on Program Comprehension*, 6 2011.
- [9] G. Cosma and M. Joy. An approach to Source-Code plagiarism detection and investigation using latent semantic analysis. *I.E.E.E. transactions on computers/IEEE transactions on computers*, 61(3):379–394, 3 2012.
- [10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.
- [11] S. Dou, J. Shan, H. Jia, W. Deng, Z. Xi, W. He, Y. Wu, T. Gui, Y. Liu, and X. Huang. Towards understanding the capability of large language models on code clone detection: A survey, 2023.
- [12] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. *ICSM*, 1 1999.

- [13] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical software engineering*, 13(6):601–643, 7 2008.
- [14] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.
- [15] Y. Gao, Z. Wang, S. Liu, L. Yang, W. Sang, and Y. Cai. TECCD: A Tree Embedding Approach for Code Clone Detection. *International Conference on Software Maintenance (ICSM)*, 9 2019.
- [16] M. Hammad, [U+FFFF] Babur, H. A. Basit, and M. Van Den Brand. Clone-Seeker: Effective code clone search using annotations. *IEEE access*, 10:11696–11713, 1 2022.
- [17] M. Harman. The role of Artificial Intelligence in Software Engineering. *International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, 6 2012.
- [18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. *Proceedings/Proceedings - International Conference on Software Engineering*, 5 2007.
- [19] H. Jin, Z. Cui, S. Liu, and L. Zheng. Improving Code Clone Detection Accuracy and Efficiency based on Code Complexity Analysis. *2022 9th International Conference on Dependable Systems and Their Applications (DSA)*, 8 2022.
- [20] B. Joshi, P. Budhathoki, W. L. Woon, and D. Svetinovic. *Software clone detection using clustering approach*. 1 2015.
- [21] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
- [22] I. Keivanloo, F. Zhang, and Y. Zou. Threshold-free code clone detection for a large-scale heterogeneous Java repository. *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 3 2015.
- [23] R. Koschke. Survey of research on software clones. Dagstuhl report, Schloss Dagstuhl, 2007.
- [24] G. Li, Y. Wu, C. K. Roy, J. Sun, X. Peng, N. Zhan, B. Hu, and J. Ma. SAGA: Efficient and Large-Scale Detection of Near-Miss Clones with GPU Acceleration. *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2 2020.
- [25] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. CCLearner: A Deep Learning-Based Clone Detection Approach. *International Conference on Software Maintenance (ICSM)*, 9 2017.
- [26] N. Li, M. Shen, S. Li, L. Zhang, and Z. Li. *STVSM: similar structural code detection based on AST and VSM*. 1 2012.
- [27] Q. Li and C. Zhang. Research on algorithm of program code similarity detection. In *2017 International Conference on Computer Systems, Electronics and Control (ICCSEC)*. IEEE, 2017.

- [28] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE transactions on software engineering*, 32(3):176–192, 3 2006.
- [29] N. Mayrand, N. Leblanc, and N. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. 1 1996.
- [30] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto. Gapped code clone detection with lightweight source code analysis. *IEEE Workshop on Program Comprehension*, 5 2013.
- [31] W. Pengcheng, S. Jeffrey, W. Yanzhao, X. Yun, and K. R. Chanchal. CCAliigner: a token based Large-Gap Clone Detector. *IEEE Conference Proceedings*, 2018:1066–1077, 1 2018.
- [32] C. Ragkhitwetsagul and J. Krinke. Siamese: scalable and incremental code clone search via multiple code representations. *Empirical software engineering*, 24(4):2236–2284, 3 2019.
- [33] C. Ragkhitwetsagul, J. Krinke, and D. Clark. A comparison of code similarity analysers. *Empirical software engineering*, 23(4):2464–2519, 10 2017.
- [34] D. Rattan, R. Bhatia, and M. Singh. Software clone detection: A systematic review. *Information and software technology*, 55(7):1165–1199, 7 2013.
- [35] C. K. Roy and J. R. Cordy. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. *IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW*, 1 2009.
- [36] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 5 2009.
- [37] T. Sager, A. Bernstein, M. Pinzger, and C. Kiefer. Detecting similar Java classes using tree algorithms. *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, 5 2006.
- [38] V. Saini, H. Sajnani, J. Kim, and C. Lopes. SourcererCC and SourcererCC-I: tools to detect clones in batch mode and during software development. *International Conference on Software Engineering - Companion, ICSE, Companion*, pages 597–600, 5 2016.
- [39] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: scaling code clone detection to big-code. *arXiv (Cornell University)*, pages 1157–1168, 5 2016.
- [40] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 11 1975.
- [41] T. Schmorleiz and R. Lämmel. Similarity management of ‘cloned and owned’ variants. *SAC '16: Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 4 2016.
- [42] G. M. Selim, K. C. Foo, and Y. Zou. Enhancing Source-Based Clone Detection Using Intermediate Representation. *Working Conference on Reverse Engineering*, 10 2010.

- [43] G. Shobha, A. Rana, V. Kansal, and S. Tanwar. *Code Clone Detection—A Systematic Review*. 1 2021.
- [44] J. Svajlenko, I. Keivanloo, and C. K. Roy. Big data clone detection using classical detectors: An exploratory study. *Journal of Software*, 27(6):430–464, 2014.
- [45] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.
- [46] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. *IEEE International Conference on Automated Software Engineering (ASE)*, 8 2016.
- [47] M. Wu, P. Wang, K. Yin, H. Cheng, Y. Xu, and C. K. Roy. LVMapper: a Large-Variance Clone Detector using sequencing alignment approach. *IEEE access*, 8:27986–27997, 1 2020.
- [48] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue. Measuring similarity of large software systems based on source code correspondence. In *Lecture Notes in Computer Science*, pages 530–544. Springer, 2005.
- [49] Y. Yu, Z. Huang, G. Shen, W. Li, and Y. Shao. ASTENS-BWA: Searching partial syntactic similar regions between source code fragments via AST-based encoded sequence alignment. *Science of computer programming*, 222:102839, 10 2022.
- [50] F. Zhang, Y. C. Jhi, D. Wu, P. Liu, and S. Zhu. A first step towards algorithm plagiarism detection. In *Proceedings of the Conference at Penn State*. Penn State, 2012.
- [51] Y. Zhang and T. Wang. CCEyes: An Effective Tool for Code Clone Detection on Large-Scale Open Source Repositories. *Information Communication and Software Engineering (ICICSE), IEEE International Conference on*, 3 2021.