COSC 3P99 Report

Name:- Vivek Salwan
Student ID:- 6951826

1. INTRODUCTION

In this report, I will explain everything that I have learned over time in my COSC 3P99 course under the guidance of Professor Naser Ezzati-Jivan. This report will cover all the research papers and articles that I have read during the course, and I will explain in detail what I have learned from them. Furthermore, I will explain the project that I have been working on, which uses large language models to generate documentation for coding projects.

2. RESEARCH PAPERS AND ARTICLES

1st - Augmenting Intelligent Document Processing (IDP) Workflows with Contemporary Large Language Models (LLMs)

The paper "Augmenting Intelligent Document Processing (IDP) Workflows with Contemporary Large Language Models (LLMs)" explores the integration of Large Language Models (LLMs) into the Intelligent Document Processing (IDP) workflow" by author Shreekant Mandvikar. This paper talks about the challenges faced by the growing volume of documents and company data that an organization needs to manage in order to function properly. This research paper suggests that the integration of LLMs into IDP workflows can significantly improve the extraction of data from these documents.

This paper has provided a really good introduction to LLMS as it talks about how LLMs are predominantly built on a class of deep learning architectures called transformer networks. It also talks about how LLMs work as explained below:-

2.1. How Large Language Model Works

Large Language Models (LLMs) are predicated on statistical methodologies that encode a probabilistic relationship among sequences of words [4], [5]. These models use a probability distribution denoted as P(w1,...,wL), which aims to approximate the empirical distribution observed in a substantial corpus of text in a specific language. The simplest form of such a model is the "1-gram" model, as given below, which operates under the assumption that words are independently distributed [6], [7].

$$P(w_1, \ldots, w_L) = \prod_{i=1}^{L} P(w_i)\,; \quad P(W) = \frac{n(w)}{W}$$

Where n(w) and W represent the number of occurrences of w in the corpus and the total number of words in the corpus.

Where the probability of a word sequence P(w1,..., wL) is computed as the product of the probabilities of individual words P(wi), the individual word probabilities are determined by the frequency of each expression in the corpus relative to the total number of words in that corpus [8], [9]. To assess the effectiveness of a language model, the standard metric employed is cross entropy, which quantifies how well the model's probability distribution mirrors the empirical distribution observed in the corpus [10], [11]—the cross-entropy.

L is calculated as a sum of logarithmic terms related to the conditional probabilities of word sequences, often denoted as:

$$L = -\frac{1}{N}\sum_{i=1}^{N-n} \log P\,(W_{i+n} \mid W_i, W_{i+1}, \ldots, W_{i+n-1})$$

Where the perplexity is represented as exp(−L). The equation is an objective function to minimise during the training phase. Various optimization techniques, such as backpropagation, can be employed.

In this paper, we can also see a detailed comparison among traditional Machine Learning (ML), Deep Learning (DL), and Large Language Models (LLMs) based on their training size data, model complexity, performance, cost, adaptability, etc. This information provides us with a lot of insights into how different models are suitable for different tasks and how LLMs are efficient at certain tasks like extracting meaningful information etc.

Table 1. Comparison among traditional Machine Learning (ML), Deep Learning (DL), and Large Language Models (LLMs)

Comparison	Traditional ML	Deep Learning	Large Language Models
Training Data Size	Large (Thousands to Millions)	Large (Thousands to Millions)	Very Large (Billions+)
Feature Engineering	Manual (Domain expertise features)	Automatic (Self-learns features)	Automatic (Self-learns required)
Model Complexity	Limited (Linear, Tree-based)	Complex (Convolutional, Recurrent)	Very Complex (Up to billions of parameters)
Interpretability	Good (Transparent algorithms)	Poor (Black-box nature)	Poorer (Extremely complex, black-box nature
Performance	Moderate (Sufficient for simpler tasks)	High (Effective for complex for multiple tasks)	Highest (State-of-the-art tasks)
Hardware Requirements	Low (CPUs sufficient)	High (GPUs often required)	Very High (Multiple GPUs, TPUs often required)
Computational Cost	Low to Moderate	High	Very High
Real-Time Capabilities	Often Suitable	Less Suitable (due to Complexity)	Generally Not Suitable
Adaptability	Lower (Fine-tuning often leaning)	Moderate (Some transfer learning)	High (Very adaptable with needed)
Software Libraries	Scikit-learn, Stats models	TensorFlow, PyTorch	Hugging Face Transformers, GPT-specific implementations

Furthermore, it also talks about the different classes of LLMs, each with their unique capabilities for different use cases, as shown below:-

Table 2. Overview of different classes of large language models

Model Class	Short Description	Typical Use Cases	Example Models
Autoregressive	Specialized in generating coherent and contextually relevant text.	Content creation, Conversational agents	GPT-3 (Generative Pretrained Transformer 3)
Encoder-only	Optimized for understanding and representing text rather than generating it.	Text classification, Sentiment analysis	BERT (Bidirectional Encoder Representations from Transformers)
Encoder-Decoder	Capable of both understanding and generating text, offering versatility.	Machine translation, Summarization	T5 (Text-to-Text Transformer), BART (Bidirectional and Autoregressive Transformers)
Specialized	Designed for specific tasks that require high performance but lower computational costs.	Efficient text classification, Information retrieval	ELECTRA (Efficiently Learning an Encoder that Classifies Token Replacements Accurately)

This table shows Autoregressive models, such as GPT-3 (Generative Pretrained Transformer 3), are compelling at generating human-like text, making them ideal for tasks such as content creation and another class Encoder-only models like BERT (Bidirectional Encoder Representations from Transformers), are designed to understand and represent textual data rather than generate it. A third class comprises Encoder-Decoder models like T5 (Text-to-Text Transformer) or BART (Bidirectional and Auto-Regressive Transformers), which are versatile in both understanding and generating text. Moreover, specialized models like ELECTRA (Efficiently Learning an Encoder that Classifies Token Replacements Accurately) are designed for efficiency and are suitable for tasks that demand lower computational resources but require high performance.

Below we can explain the experiment in 4 stages

3.1 Document Classification Stage

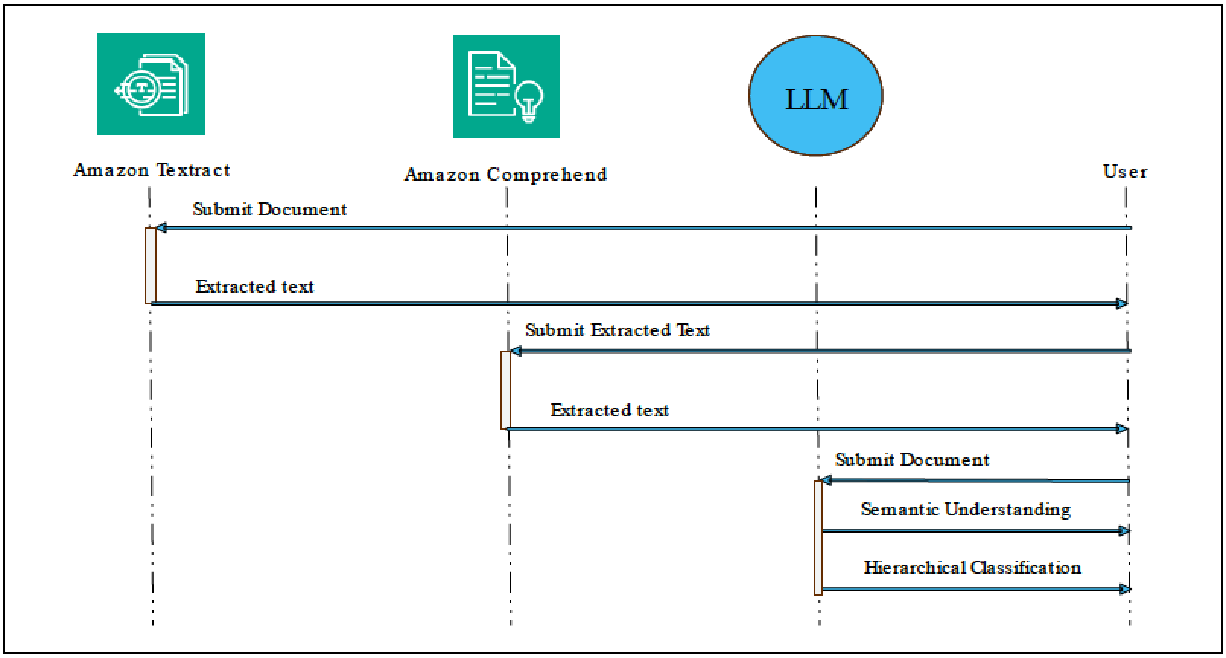


Fig. 1 Contributions of LLMs in the classification stage of IDP

In this stage, we make use of Amazon Web Services (AWS) as AWS provides Amazon Textract and Amazon Comprehend. Amazon Textract helps extract text and other data from scanned documents. In the first stage, the user needs to submit a document to Amazon Textract, which begins the IDP process. After extracting text and data, this includes recognizing text from various document formats and layouts. It sends the extracted text back to the user for the next service in the workflow. The extracted document is then submitted to Amazon Comprehend, which uses Natural Language Processing (NLP) techniques to analyze the context and semantic details of the extracted text, and in parallel, the extracted text is also submitted to an LLM. The LLM employs advanced techniques for a more nuanced document classification like using Semantic understanding and Hierarchical classification. In the end, the LLM sends the processed document to the user, concluding its role in the process. The output includes a more refined categorization of the document.

3.2 Document Extraction Stage

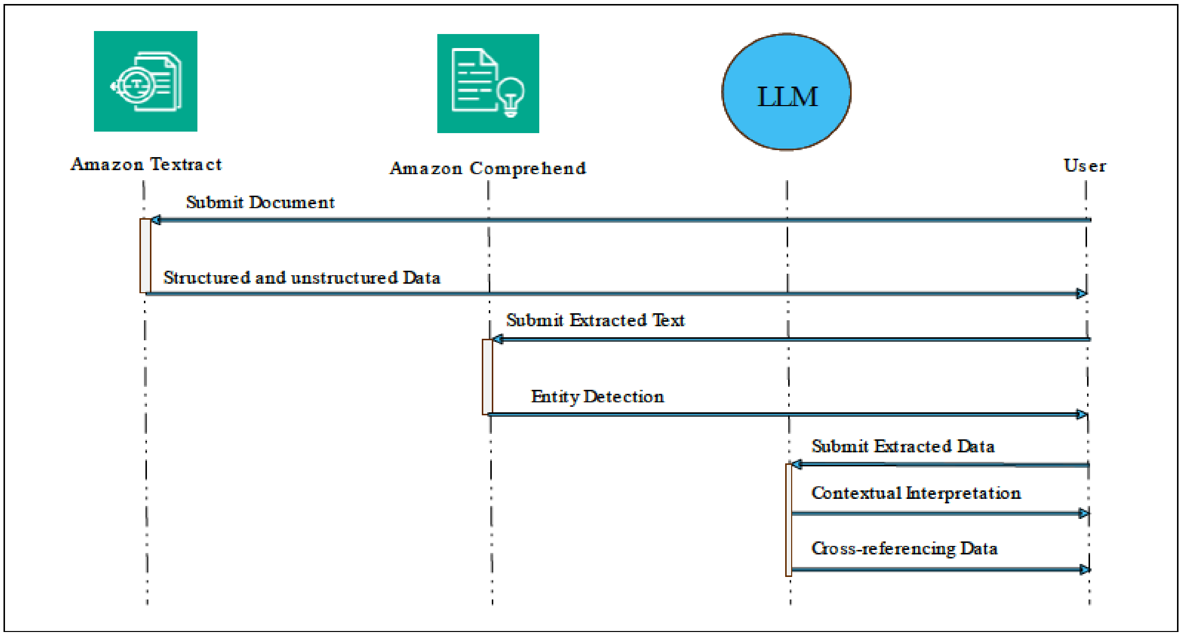


Fig. 2 Contributions of Large Language Models (LLMs) in the extraction stage of IDP

In this stage, we perform all the same steps as the previous stage as you can see in the above figure but when we feed the data into the LLM we improve it by using contextual interpretation to improve the understanding of extracted data by analyzing its context and we use Cross-reference data as LLMs link related data across the document ensuring consistent interpretation and reducing errors in data extraction. This stage is important for transforming raw data into structured formats that can be easily used in databases or other applications.

3.3 Review and Validation Stage

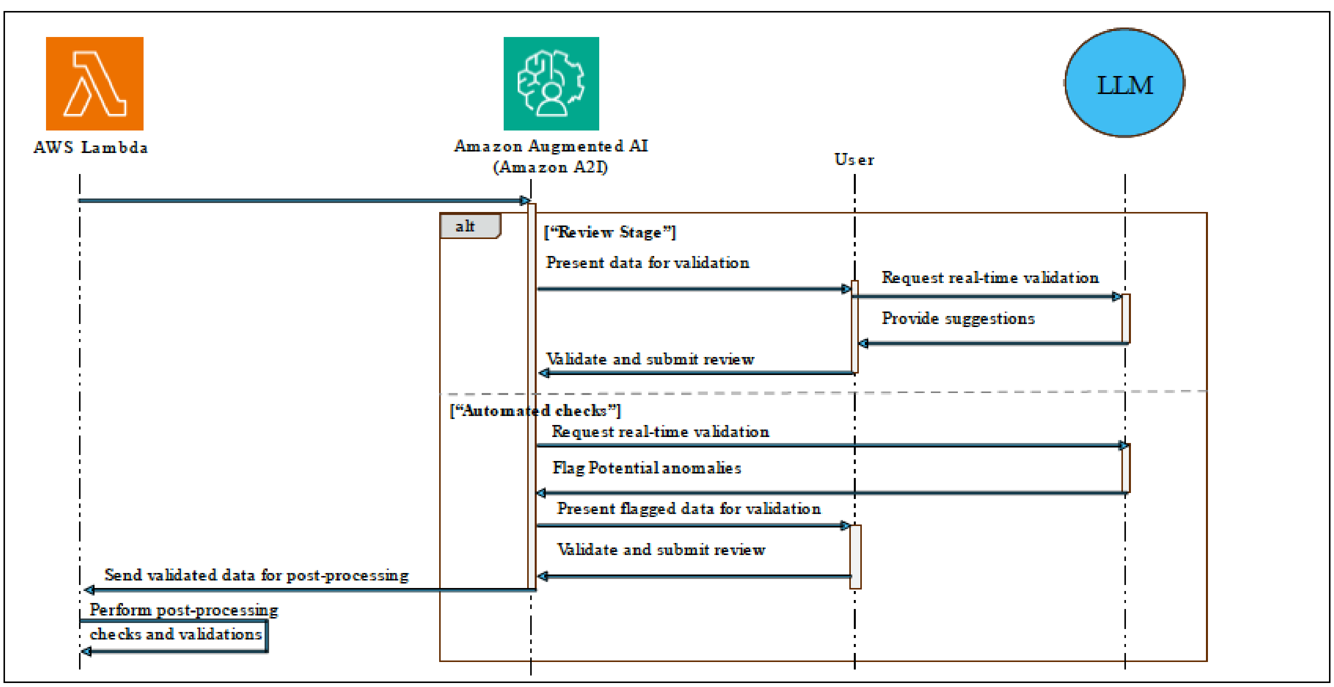


Fig. 3 Contributions of Large Language Models (LLMs) in the review/validation stage of IDP

In this stage, review and validation take place where the process begins with AWS Lambda sending the extracted data to Amazon Augmented AI (A2I) for review. Amazon A2I facilitates the integration of human reviewers in the workflow. In this process, Human reviewers assess the extracted data to validate its accuracy and completeness. In parallel to the manual review, LLMs are also used for automated anomaly detection. LLMs analyze the data for any inconsistencies or unusual patterns that may indicate errors. Once the review process is complete, the validated data is sent back to AWS Lambda for post-processing. The final step involves AWS Lambda finalizing the data processing, incorporating all the corrections and validations from the review stage. The data is now ready for further use in applications or for entry into databases.

3.4 Document Enrichment Stage

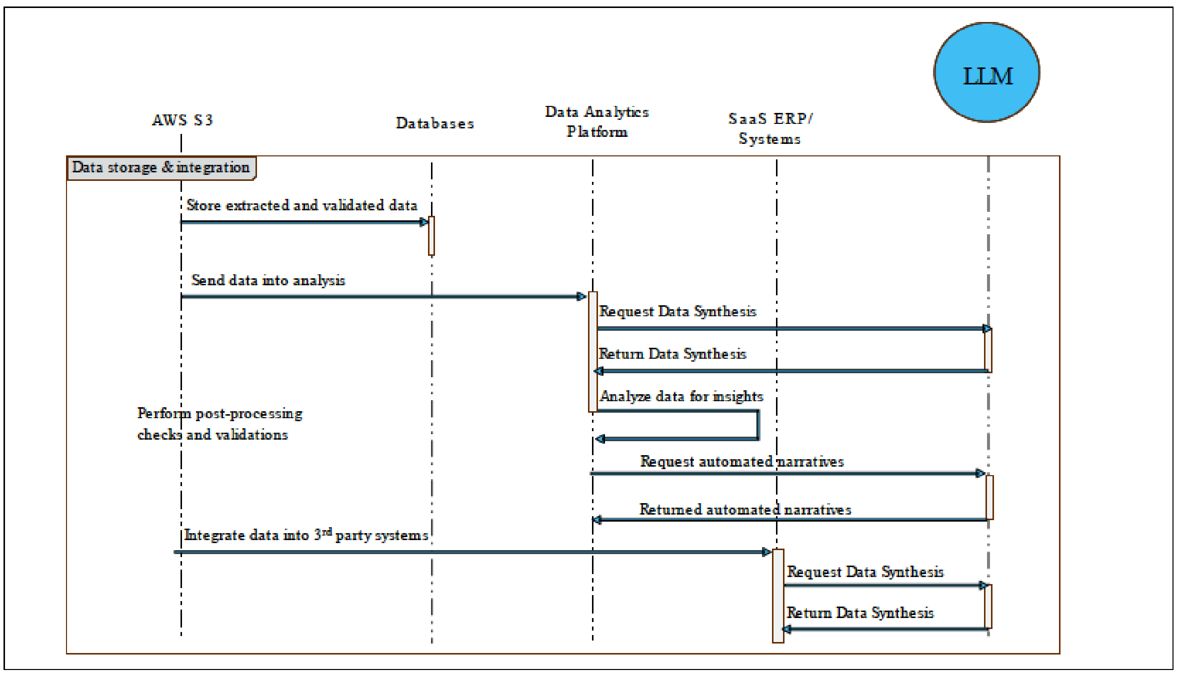


Fig. 4 Contributions of Large Language Models (LLMs) in the review/validation stage of IDP

In the final stage, After the data has been extracted, validated, and possibly enriched in previous stages, it is stored in Amazon Simple Storage Service (S3) and possibly other databases. LLMs are utilized to synthesize the stored data to ensure consistency and coherence across different data types and sources. The synthesized data is then sent to Data Analytics platforms, where it is prepared for detailed analysis. As part of making the data more accessible and understandable, LLMs are employed to generate automated narratives or summaries based on the analyzed data. Following analysis, the data is integrated into third-party applications or systems, such as Enterprise Resource Planning (ERP) systems or other Software as a Service (SaaS) platforms. Once integrated, the data is ready for use within the organization's broader ecosystem, supporting various business functions, from customer relationship management to financial analysis and beyond.

In conclusion, this research paper effectively explains how Large Language Models (LLMs) can be integrated into the Intelligent Document Processing (IDP) workflow to enhance each stage, from classification and extraction to review, validation, and integration. By using LLMs large organizations can significantly improve their methods of document processing. LLMs provide advanced capabilities such as semantic understanding, hierarchical classification, contextual interpretation, and automated narrative generation, which exceed traditional document processing methods. However, we should also consider that the implementation of LLMs also introduces complexities related to computational demands, potential biases, and the need for sophisticated model management. Organizations must consider all these factors to fully leverage the potential of LLMs in enhancing IDP workflows.

2nd - Automatic Code Documentation Generation Using GPT-3

The research paper "Automatic Code Documentation Generation Using GPT-3" by Junaed Younus Khan and Gias Uddin explores the use of OpenAI's Codex, a GPT-3-based model, for generating code documentation automatically. Codex is trained on both natural and programming languages and shows promise in improving the efficiency of software development by automating the documentation process.

The paper begins by addressing the importance of well-maintained code documentation in software development, especially highlighted during the shift to remote work due to COVID-19. Traditional documentation practices are costly, often outdated, and unpopular among developers, prompting a need for automation. Here we use Codex, a model derived from GPT-3 that is specially tuned for understanding and generating programming content and is trained on a vast dataset that includes GitHub contributions across multiple programming languages. This paper talks about an experiment that is carried out using Codex which is explained below:-

1. Data Collection and Preprocessing

Automatic Code Documentation Generation Using GPT-3

Table 1: Statistics of CodeSearchNet [25]

Language	Train	Valid	Test
Java	164,923	5,183	10,955
Python	251,820	13,914	14,918
PHP	241,241	12,982	14,014
GO	167,288	7,325	8,122
JavaScript	58,025	3,885	3,291
Ruby	24,927	1,400	1,261

The researchers used the CodeSearchNet dataset, which is widely employed for various software engineering tasks involving code understanding and generation. This dataset includes pairs of code and corresponding documentation for six programming languages: Java, Python, PHP, GO, JavaScript, and Ruby. To perform the experiment properly the preprocessing steps include Removing comments from the code to ensure that the model's output was purely based on the code's content rather than comments. Filtering out examples where the code could not be parsed into an abstract syntax tree, the documentation was too short or too long or contained non-English language or special tokens like <img> tags.

2. GPT-3 Model and Parameter Setup

```
Code:
    def add(x, y):
        return x+y
Documentation: Adds two numbers.
Code:
    def subtract(x, y):
        return x-y
Documentation: [To be generated by Codex]
```

Figure 2: Sample prompt format for one-shot learning

The Codex model, a variant of GPT-3 specifically designed for understanding and generating code, was utilized. This model is pre-trained on a large corpus of natural language and programming code from sources like GitHub. The model was tested with zero-shot and one-shot learning approaches. In zero-shot learning, the model received only a task description without any examples. In one-shot learning, it received one example of a code-documentation pair before being asked to generate documentation for a new piece of code. Parameters like 'Temperature' and 'Top-p' were adjusted to control the randomness of the generated output, with 'Temperature' set low to reduce randomness and ensure more predictable outputs.

3. Evaluation of Generated Documentation

Table 2: Results on documentation generation (BLEU score)

Model	Ruby	JavaScript	GO	Python	Java	PHP	Overall
Seq2Seq [48]	9.64	10.21	13.98	15.93	15.09	21.08	14.32
Transformer [54]	11.18	11.59	16.38	15.81	16.26	22.12	15.56
RoBERTa [31]	11.17	11.90	17.72	18.14	16.47	24.02	16.57
CodeBERT [17]	12.16	14.90	18.07	19.06	17.65	25.16	17.83
PLBART [5]	14.11	15.56	18.91	19.30	18.45	23.58	18.32
CoTexT (2-CC) [40]	13.07	14.77	19.37	19.52	19.1	24.47	18.38
CoTexT (1-CC) [40]	14.02	14.96	18.86	19.73	19.06	24.58	18.55
REDCODER [38]	-	-	-	21.01	22.94	-	N/A
REDCODER-EXT [38]	-	-	-	20.91	22.95	-	N/A
Codex (0-shot)	5.41	9.83	15.80	18.93	13.59	13.32	12.81
Codex (1-shot)	16.04	16.58	20.94	22.28	22.81	25.13	20.63

The effectiveness of the generated documentation was measured using both quantitative and qualitative metrics:

Quantitative Metrics: BLEU Score is a standard metric for evaluating machine-generated text based on n-gram similarity between the generated text and a reference text. The study used a smoothed version of the BLEU score to account for the variability in documentation length and content. Moreover, we observed that though Codex with zero-shot learning could not achieve satisfactory results (mostly because it fails to learn the expected documentation format), the performance greatly improves with one-shot learning. Codex (with one-shot) shows the best overall performance among all approaches with an average BLEU score of 20.63 while the nearest competitor CoTexT achieves 18.55 (11.21% improvement).

Java

```
protected ImageIcon createImageIcon(String path, String description) {
    java.net.URL imgURL = getClass().getResource(path);
    if (imgURL != null) {
      return new ImageIcon(imgURL, description);
    } else {
      System.err.println("Couldn't find file: " + path);
      return null;
    }
  }
```

Actual: Returns an ImageIcon, or null if the path was invalid.
Codex: Returns an ImageIcon, or null if the path was invalid.

Python

```
def check_for_file(self, file_path):
    try:
        files = self.connection.glob(file_path, details=False,
                invalidate_cache=True)
        return len(files) == 1
    except FileNotFoundError:
        return False
```

Actual: Check if a file exists on Azure Data Lake.
Codex: Check if a file exists on the remote machine.

PHP

```
public static function base256ToDec($number)
{
    Validate::canCastAsString($number, 'number');
    $result = 0;
    $base  = 1;
    for ($i = strlen($number) - 1; $i >= 0; $i--) {
        $result = bcadd($result, bcmul(ord($number[$i]), $base));
        $base  = bcmul($base, 256);
    }
    return $result;
}
```

Actual: Convert base 256 number to decimal number.
Codex: Converts a base 256 number to a base 10 number.

JavaScript

```
function validateProtocolOptions(protocolOptions) {
  if (!protocolOptions) {
    throw new TypeError('protocolOptions not defined in options');
  }
  const version = protocolOptions.maxVersion;
  if (version && (typeof version !== 'number' ||
        !types.protocolVersion.isSupported(version))) {
    throw new TypeError(util.format('protocolOptions.maxVersion provided (%s)
          is invalid', version));
  }
}
```

Actual: Validates the protocol options.
Codex: Validate protocol options.

Ruby

```
def remove_file(*paths)
  path = File.join(*paths)
  log.debug(log_key) { "Removing file {path}" }
  FileUtils.rm_f(path)
  path
 end
```

Actual: Remove the file at the given path.
Codex: Removes the file located at the given path. If the path is not present a noop is performed.

GO

```
func (os *orderedSet) append(s string) {
 if _, ok := os.included[s]; !ok {
  os.list = append(os.list, s)
  os.included[s] = struct{}{}
 }
}
```

Actual: append adds s to the end of os, only if it is not included already.
Codex: append adds s to the orderedSet if it is not already present.

Figure 3: Examples of documentation by Codex (1-shot)

Qualitative Metrices:

Documentation length and readability is measured by the Flesch-Kincaid Grade Level, which assesses the readability based on sentence length and word complexity. Its Informativeness is evaluated using TF-IDF scores to compare the information content of the generated documentation against the actual documentation.

In conclusion, the experiment emphasizes the potential of using advanced language models, like Codex, to automate the generation of code documentation. It showcases the model's ability to adjust to different programming languages and create top-notch documentation with minimal input. This paves the way for its practical use in software development and maintenance tasks.

3rd - Enabling BLV Developers with LLM-driven Code Debugging

The research paper titled "Enabling BLV Developers with LLM-driven Code Debugging" by Clark Saben and Prashant Chandrasekar talks about BLVRUN, a command-line tool designed to assist blind and low-vision (BLV) developers by summarizing complex error messages into concise, easily understandable summaries. This tool is used to fine-tune a large language model to enhance accessibility and streamline the debugging process for developers who face significant challenges using traditional debugging tools.

Below we can explain the experiment caried out in the paper:-

Enabling BLV Developers with LLM-driven Code Debugging x '24, , USA

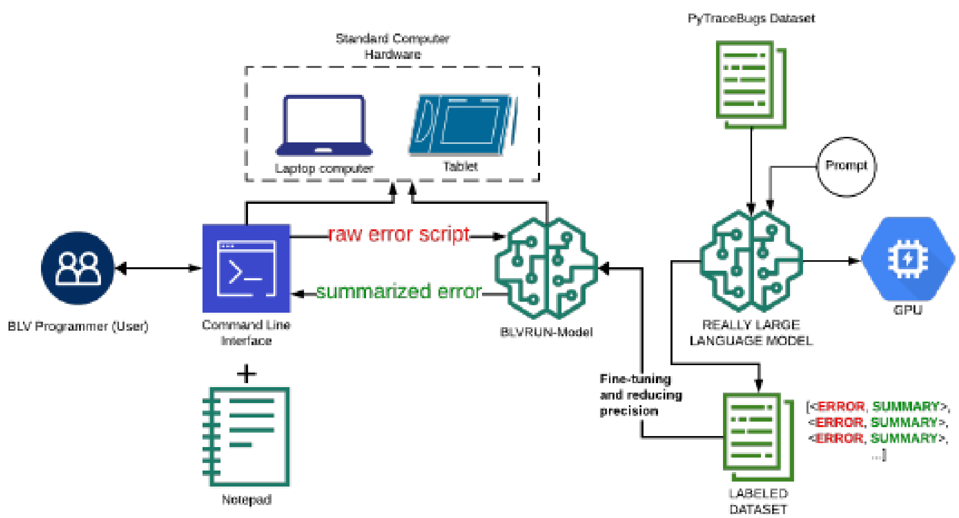


Fig. 1. Architecture and Development Components of BLVRUN. Starting from the left, a BLV programmer, who using CLI and text buffers executes their Python code. When an error is produced, BLVRUN's script captures the verbose and unstructured text and only presents the user with a concise and accurate description of the error. This is possible because BLVRUN's model is fine-tuned using a dataset we created from PyTraceBugs. Finally, BLVRUN is optimized to run on any machine, thereby not requiring BLV programmers to depends on IDEs and/or switch contexts with ChatGPT-like solutions.

This experiment focuses on evaluating the effectiveness of the BLVRUN tool, which is designed to aid blind and low-vision (BLV) developers by summarizing traceback errors into understandable summaries. This evaluation aims to assess how well BLVRUN performs in real-world debugging scenarios compared to standard models and according to specific metrics.

1. Dataset Utilization

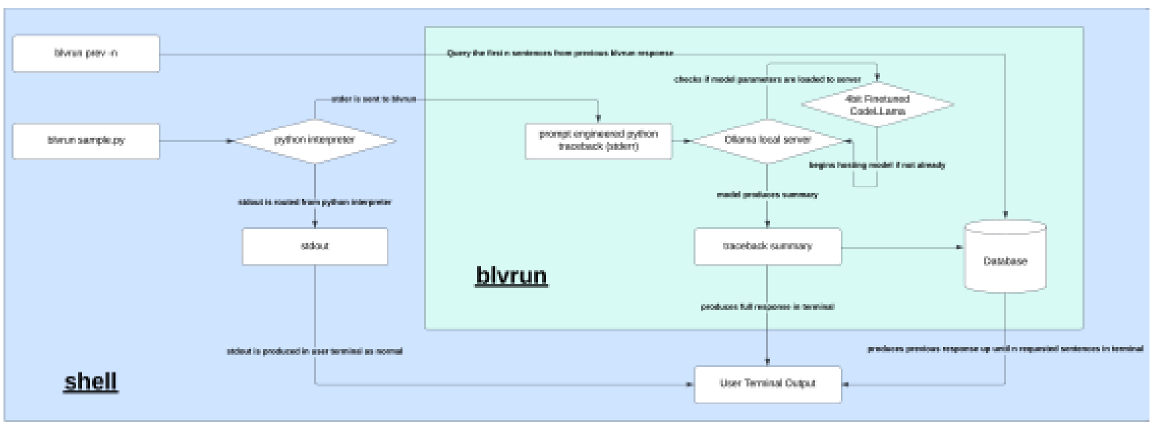
The PyTraceBugs Dataset is used for both training and testing. It includes a variety of traceback errors from Python programs, which are crucial for fine-tuning the model to accurately summarize common errors encountered by developers.

2. Fine Tuning

For fine tuning BLVRUN uses a version of the CodeLlama model, fine-tuned specifically on the PyTraceBugs dataset. This fine-tuning process adapts the model to better understand and generate summaries for Python traceback errors. Moreover, to enhance performance, the model's precision is reduced, by a process known as *quantization*. This step is crucial for ensuring that the model runs efficiently on standard consumer hardware.

3. User Interface

3.5 User interface



The above diagram explains the BLVRUN interface, When blvrun sample.py is executed in the shell, the prompt is sent to our model that is hosted on a Ollama server. Our model produces a traceback summary that is sent back to the terminal and saved in a database. BLV programmers can see previously generated summaries using the blvrun prev -n command.

4. Evaluation

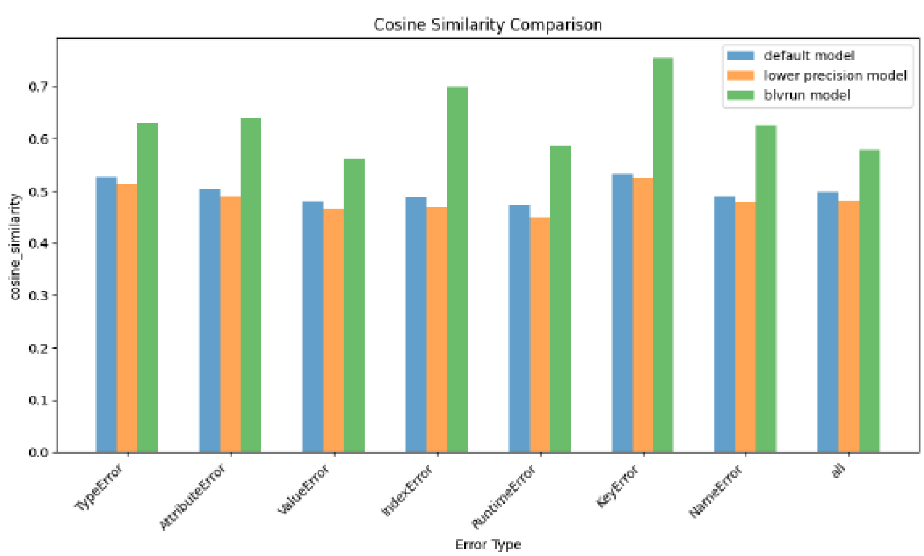


Fig. 4. Cosine similarity scores of summaries generated by (1) base model (with no fine-tuning or lowered precision), (2) base model (with lowered precision), (3) BLVRUN's fine-tuned and optimized model compared against "gold standard"

Cosine Similarity: This metric is used to evaluate the similarity between the error summaries generated by BLVRUN and a "gold standard" set of summaries. The gold standard is presumably generated using the most accurate settings or another high-performing model.

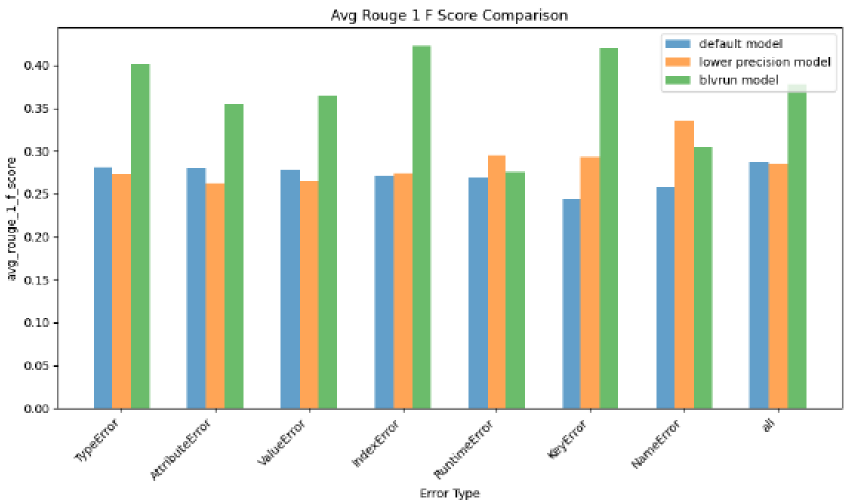


Fig. 5. ROUGE-1 f-scores of summaries generated by (1) base model (with no fine-tuning or lowered precision), (2) base model (with lowered precision), (3) BLVRUN's fine-tuned and optimized model compared against "gold standard"

ROUGE-1 Score: This metric measures the overlap of unigrams between the generated summaries and the gold standard. It assesses how many of the same words appear in both texts, which is an indicator of textual similarity and relevance.

The results, shown in figures within the paper (e.g., Figures 4 and 5), demonstrate that the fine-tuned and optimized BLVRUN model outperforms the base models in terms of both cosine similarity and ROUGE-1 scores. This indicates that the summaries it produces are both accurate and relevant, closely matching the gold standard.

In conclusion, this experiment shows that BLVRUN, with its specific optimizations and fine-tuning, provides significant benefits over standard models, making it an effective tool for helping BLV developers understand and resolve errors in their code more efficiently.

4th - CodePlan: Repository-level Coding using LLMs and Planning

The document titled "CodePlan: Repository-level Coding using LLMs and Planning" by Microsoft Research, India presents a detailed exploration of how Large Language Models (LLMs) can be effectively used to automate extensive coding tasks across entire repositories, rather than just localized coding issues. Traditional LLM applications like GitHub Copilot excel in localized coding scenarios but struggle with repository-wide tasks where changes need to be coordinated across files and may depend on each other.

Experimental Setup

1. Datasets

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet

Repositories	Migration (C#)			Temporal Edits (Python)		
	Int-1	Int-2	Ext-1	T-1	T-2	T-3
Number of files	91	168	55	21	137	4
Lines of code	8853	16476	8868	3883	20413	1874
Number of files changed	47	97	21	2	2	3
Number of seed changes	41	63	42	2	1	1
Number of derived changes	110	375	16 0	8	3	10
Size of diff (lines)	1744	4902	1024	104	15	39
Size of seed edits (lines)	242	242	379	76	4	1
Prompt template size (lines)	81	81	81	75	75	75
URL	-	-	[7]	[8]	[1]	[3]

Table 2. Dataset statistics. Int-1,2 are internal (proprietary) repositories, Ext-1 and T-1,2,3 are external (public GitHub) repositories.

Multiple repositories were chosen for the experiment to cover a range of coding tasks. These included internal proprietary repositories from a large product company and publicly available repositories from GitHub. Each repository varied in size and complexity, ensuring a robust test environment. The tasks were designed to reflect real-world software development challenges, such as migrating from one logging framework to another or making a series of code edits dictated by evolving requirements or dependency updates.

2. CodePlan Operations

CodePlan was configured with initial seed edits, which are the starting point for each task. These edits are predefined modifications that simulate typical changes a developer might make, such as updating method calls to a new API version. The system then generated a plan based on these seed edits, using its planning algorithm to determine all necessary subsequent edits throughout the repository. This planning accounted for dependencies and potential impacts across the repository's codebase. For each planned edit, CodePlan used an LLM to generate the actual code changes. The LLM was provided with specific prompts that included relevant code context, ensuring that the suggested edits were applicable and syntactically correct.

3. Evaluation Metrics

CodePlan: Repository-level Coding using LLMs and Planning

Dataset	Approach	Matched Blocks	Missed Blocks	Spurious Blocks	Diff BLEU	Levenshtein Distance	Validity Check
C# Migration Task on Internal (Proprietery) Repositories							
Int-1 (Logging)	CodePlan (Iter 1)	**151**	**0**	0	0.99	60	✗ (4)
	CodePlan (Iter 2)	**151**	**0**	0	1.00	0	✓ (0)
	Build-Repair	82	69	13	0.81	6465	✗ (46)
Int-2 (Logging)	CodePlan (Iter 1)	**438**	**0**	0	0.99	90	✗ (6)
	CodePlan (Iter 2)	**438**	**0**	0	1.00	0	✓ (0)
	Build-Repair	337	101	25	0.66	7496	✗ (68)
C# Migration Task on External (Public) Repositories							
Ext-1 (NUnit-XUnit)	CodePlan (Iter 1)	**58**	**0**	0	1.00	0	✓ (0)
	Build-Repair	52	6	1	0.94	530	✗ (8)
Python Temporal Edit Task on External (Public) Repositories							
T-1	CodePlan (Iter 1)	**8**	**2**	0	**0.90**	**1044**	✗
	Pyright-Repair	5	5	0	0.76	1090	✗
	Pyright-Strict-Repair	**8**	**2**	0	**0.90**	**1045**	✗
	Coeditor-CodePlan	**8**	**2**	0	**0.90**	1160	✗
	Coeditor-Pyright-Repair	5	5	0	0.66	1205	✗
	Coeditor-Pyright-Strict-Repair	5	5	0	0.65	1139	✗
T-2	CodePlan (Iter 1)	**4**	**0**	0	**0.86**	**248**	✓
	Pyright-Repair	1	3	0	0.00	344	✗
	Pyright-Strict-Repair	1	3	0	0.00	344	✗
	Coeditor-CodePlan (Iter 1)	3	1	0	0.82	254	✗
	Coeditor-Pyright-Repair	1	3	0	0.00	344	✗
	Coeditor-Pyright-Strict-Repair	1	3	0	0.00	344	✗
T-3	CodePlan (Iter 1)	**11**	**0**	0	**0.92**	**358**	✓
	Pyright-Repair	1	10	0	0.00	798	✗
	Pyright-Strict-Repair	1	10	0	0.00	798	✗
	Coeditor-CodePlan (Iter 1)	10	1	0	0.78	717	✗
	Coeditor-Pyright-Repair	1	10	0	0.00	798	✗
	Coeditor-Pyright-Strict-Repair	1	10	0	0.00	798	✗

Table 3. Comparison of CodePlan's repository edit metrics with Build-Repair baseline. Higher values of Matched Blocks, Diff BLEU, and lower values of Missed Blocks, Spurious Blocks, Levenshtein Distances are better.

The primary metrics for evaluation were the correctness and completeness of the code changes. Correctness was measured by the ability of the repositories to compile and pass tests after the edits. Completeness was assessed by comparing the final state of the repository against a ground truth of expected outcomes after all edits were applied. CodePlan's performance was compared against baseline methods that did not use the planning approach. These baseline methods were typical LLM-driven code editors that handled each edit independently without considering broader repository-level impacts.

4. Results and Analysis

CodePlan demonstrated a superior ability to handle complex coding tasks correctly and completely across repositories. It successfully managed dependencies and interactions between different parts of the codebase, significantly reducing errors compared to the baselines.

The experiment also tested CodePlan's adaptability by introducing changes in the tasks and observing how well the system adjusted its planning and execution strategy. This included handling unexpected changes and errors introduced during the coding process.

In Conclusion, the results of the experiment validated the efficacy of using a planning-based approach with LLM integration for managing large-scale, complex coding tasks in software repositories. CodePlan not only outperformed traditional LLM-based code editors but also demonstrated its potential to significantly automate and improve the software development lifecycle. This experiment highlights the advantages of combining AI-driven planning with code generation technology to enhance productivity and accuracy in software engineering tasks.

Workshops: -

Along with studying research paper I have also attended different sessions like:-

This session was hosted by the computer science club at Brock university. I found the engineering session on promoting to be very informative. We focused on the basics of promoting and had a hands-on experience. We learned about the history of promoting, how to use them effectively, and how to create prompts that achieve the desired results. During the session, we used different LLMs to craft prompts and get various answers. We then compared our answers to check the accuracy of each other's promotes. The session also covered different types of promoting techniques like one-shot and two-shot promoting, which were used in some of the research papers I have mentioned above.

PROJECT

For this project, I have created four different classes that interact with OpenAI's GPT-3 Model to perform different tasks like parsing and it generates a basic level of documentation based on the given input and stores its output in an Excel file. below I have explained the functionality of the different classes:-

The Parser Class:- is used for parsing Java source files and extracting information about classes and methods for documentation generation. It utilizes the Java Compiler API to process Java files and the Java Compiler Tree API to traverse the abstract syntax tree (AST) of the parsed files. The extracted information is then used to generate documentation in CSV format.

The OpenAI Class:- is used controller for interacting with the OpenAI API. It is annotated with @RestController, indicating that it handles web requests. The class includes methods for querying the OpenAI model with prompts and retrieving the responses as JSON objects. The OpenAI model is utilized to generate text completions or responses based on the provided prompts.

The Main Class:- serves as the entry point for the application. It contains the main method, which initializes and runs various components of the project. This includes creating new instances of the Parser class to parse Java source files and interacting with the OpenAI API to query the AI model.

The documentation generator Class:- is used to generate CSV documentation from provided data. Its generateCSV() method accepts a list of String arrays, formats them as comma-separated lines, and writes them to a file named "documentation.csv". It handles file writing and error handling internally, providing a streamlined approach for generating documentation.

The following images illustrate the input code (Sample3.java) and the corresponding output stored in the CSV file for our program. We have included some sample test codes in the sampleTestCodes folder which we used to test the program.

	A	B	C	D	E
1	Class Name	Method Name	Return Type	Parameters	
2	WebController	<init>	Constructor	Bart bart	
3	WebController	register	String		
4	WebController	signIn	String		
5	WebController	getSummary	String	@RequestParam(value = "input")	
6	String input	Model model			
7	WebController	createUser	String	@ModelAttribute	
8	User user	HttpSession session			
9	WebController	isValidURL	boolean	String urlStr	
10					
11					
12					
13					

Main.java OpenAi.java Sample1.java Sample2.java Sample3.java Parser.java DocumentationGenerator.java

```java
 * Controller for web-related actions.
 */

@Controller
public class WebController {


    @Autowired
    private final Bart bart;


    @Autowired
    private UserServiceImpl userService;

    private static final Logger logger = Logger.getLogger(Bart.class.getName());

    /**
     * Constructor for WebController.
     *
     * @param bart The Bart instance to use.
     */

    public WebController(Bart bart) { this.bart = bart; }

    /**
     * Endpoint for user registration.
     *
     * @return The name of the view to render.
     */

    @GetMapping("/register")
    public String register() { return "index"; }

    /**
     * Endpoint for user sign in.
     *
     * @return The name of the view to render.
     */

    @GetMapping("/signin")
    public String signIn() { return "index"; }

    /**
     * Endpoint for getting a summary.
     *
     * @param input The input from the user.
     * @param model The model to use.
     * @return The name of the view to render.
     * @throws IOException If an I/O error occurs.
     */
```

The above image shows the input class we used and the image of the excel file shows us how's the required output is generated and stored based on the contents of the file.

The code for the project is available in the GitHub repository.

REFERENCES

[1] Mandvikar, S. (2023). Augmenting intelligent document processing (IDP) workflows with contemporary large language models (llms). International Journal of Computer Trends and Technology, 71(10), 80–91. https://doi.org/10.14445/22312803/ijctt-v71i10p110

[2] Khan, J. Y., & Uddin, G. (2022). Automatic code documentation generation using GPT-3. Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. https://doi.org/10.1145/3551349.3559548

[3] C. Saben and P. Chandrasekar, "Enabling BLV Developers with LLM-driven Code Debugging," in Proceedings of the x '24, ACM, New York, NY, USA, 2018, pp. 1-8. DOI: XXXXXXX.XXXXXXX.

[4] R. Bairi, A. Sonwane, A. Kanade, V. D. C., A. Iyer, S. Parthasarathy, S. Rajamani, B. Ashok, and S. Shet, "CodePlan: Repository-level Coding using LLMs and Planning," in Proceedings of the x '24, ACM, New York, NY, USA, 2023, pp. 1-8. DOI: 10.48550/arXiv.2309.12499.

[5] OpenAI. "ChatGPT: Fine-Tuned from GPT-3.5 for Dialogue." In Proceedings of the x '24, ACM, New York, NY, USA, 2024, pp. 1-8. DOI: 10.48550/arXiv.2405.67890.

[6] Grammarly. "Free AI Writing Assistance." In Proceedings of the x '24, ACM, New York, NY, USA, 2024, pp. 1-8. DOI: 10.48550/arXiv.2405.67890.