
Join us for a crash-course on GenAI
featuring a series of talks and workshops
by Google, PwC, Microsoft, Deloitte,
and more.
KEYNOTE SPEAKER
ChatGPT for Job-Seekers:
The Right Way to Use AI to
Accelerate Your Search
Jeremy Schifeling
Founder, The Job Insiders
Ex-Google and Ex-LinkedIn
9:30 AM to 3:10 PM | Friday, December 6th | Ross School of Business
Join the case competition upon registration to secure a free
ticket and a chance to win $500!
Register by November 26th for early bird pricing!
Breakfast and Zingerman's lunch included
Powered by
MICHIGAN ROSS
BUSINESS+TECH
Sponsored by
ROSS
Career Development Office
gm
general motors

Artificial Intelligence for Software Engineering (AI for SE)

One-Slide Summary

- **Foundations of Data Science:** The theoretical basis of today's artificial intelligence systems and applications is built upon the foundations of data science, encompassing key concepts from statistics, operations research, machine learning, and computer science.
- **AI for SE:** Artificial Intelligence significantly enhances software engineering by automating and improving various aspects of the development process and maintaining code quality, making it a versatile tool for modern software development.
- **LLMs:** Large Language Models (LLMs) are currently the most successful AI techniques in software engineering due to their ability to understand and generate human-like text, which bridges the gap between natural language and programming languages.

Learning Objectives: by the end of today's lecture, you should be able to…

1. (*Knowledge*) describe the primary activities in software engineering using AI

2. (*Value*) understand why the applications of AI in software engineering are important

3. (*Skill*) Review some recent papers

Overview

- Background
- What is Artificial Intelligence (AI) for Software Engineering (SE)?
- What are the applications of AI in SE?
  - Can LLMs help us do program verification?
  - Can LLMs help us write unit tests?
  - Can LLMs help us do mutation testing?
  - Can LLMs help us do vulnerability analysis?
  - Can LLMs help us fix bugs and write new code?
  - Can LLMs help us do test generation?

Background

A Brief History of AI

- Artificial Intelligence (AI) has a rich history that dates back to the mid-20th century. The field was officially founded in 1956 during the Dartmouth Conference, where the term "artificial intelligence" was coined by John McCarthy.
- Early AI research focused on symbolic methods and problem-solving. However, progress was slow due to limited computing power and data.
- The 1980s saw a surge in AI interest, known as the first AI summer, driven by expert systems. This was followed by an AI winter in the late 1980s and early 1990s, a period of reduced funding and interest due to unmet expectations.

AI Summer from 2010-2017

- The resurgence of AI began in the 2000s with the advent of machine learning and the availability of big data.
- The development of deep learning techniques in the 2010s, particularly neural networks, marked another AI summer, leading to significant breakthroughs in image and speech recognition.

Statistical Models to Neural Networks

- The 2000s saw the rise of statistical models trained on large datasets, leveraging the increasing availability of internet data.
- In 2009, most NLP tasks used statistical language models as they could usefully ingest large datasets.
- Neural networks began to dominate NLP tasks around 2012, using dense vector representations of words.

What is Deep Learning?

- The concept of deep learning typically involves the use of deep neural networks.
- Deep learning is a subset of machine learning that uses neural networks with multiple layers (hence "deep") to model complex patterns in data.
- These deep neural networks are designed to simulate the way the human brain processes information, allowing them to perform tasks such as image and speech recognition, natural language processing, and more.
  - https://www.ibm.com/topics/deep-learning
  - https://builtin.com/machine-learning/deep-learning

Deep Neural Networks (DNNs)

- Deep neural networks (DNNs) are a type of artificial neural network with multiple layers between the input and output layers.
- These layers allow the network to learn and model complex patterns and relationships within data.
- Each layer extracts increasingly abstract features from the input, enabling the network to perform tasks such as image and speech recognition, natural language processing, and more accurately.

AI Spring or AI Golden Age: Since 2017

- Many experts consider the period starting around 2017 as the beginning of a new "AI spring" or "golden age" of AI.
- This era is characterized by rapid advancements in deep learning, significant improvements in computational power, and the widespread availability of big data.
- These factors have led to breakthroughs in various AI applications, such as natural language processing, computer vision, and autonomous systems.
- https://knowledge.wharton.upenn.edu/article/ai-entering-golden-age/

Why DNNs are so dominant in today's AI applications?

- DNNs have become dominant in today's AI systems due to their superior performance and versatility.
- They can handle large datasets and complex models, making them suitable for various applications.
- Advances in computational power, such as GPUs, and algorithm improvements have enhanced their efficiency and effectiveness.
- This combination of factors has made DNNs a cornerstone of modern AI, driving significant advancements across various industries.

The Nobel Prize in Physics 2024

The Nobel Prize in Physics 2024 was awarded jointly to John J. Hopfield and Geoffrey E. Hinton "for foundational discoveries and inventions that enable machine learning with artificial neural networks"

Ill. Niklas Elmehed © Nobel Prize Outreach

John J. Hopfield

Prize share: 1/2

Ill. Niklas Elmehed © Nobel Prize Outreach

Geoffrey Hinton

Prize share: 1/2

The Transformer Era (2017- Present)

- The introduction of the transformer model using deep neural network (DNN) architecture in 2017 revolutionized the NLP field.
- At the 2017 NeurIPS conference, Google researchers introduced the transformer architecture in their landmark paper "Attention Is All You Need," which could handle long-range dependencies more efficiently than RNNs and LSTMs.

- https://web.eecs.umich.edu/~movaghar/Attention All You Need 2017.pdf

Pre-trained Transformers

- Transformers led to the development of powerful LLMs like BERT (2018), which focuses on understanding context, and GPT-3 (2020), which excels at generating coherent and contextually relevant text.
- These models, like GPT (Generative Pre-trained Transformer), have significantly advanced the field of natural language processing (NLP) and AI.

The Current State of AI

- Artificial intelligence's influence on society has never been more pronounced.
- Since ChatGPT became a ubiquitous feature on computer desktops in late 2022, the rapid development and deployment of generative AI and large language model (LLM) tools have started to transform industries and show the potential to touch many aspects of modern life.

https://www.weforum.org/stories/2024/04/stanford-university-ai-index-report/

Large Language Models (LLMs)

- Large Language Models(LLMs) use deep learning techniques, particularly transformer architectures, to process and generate text. Some well-known examples include OpenAI's GPT series (GPT-3, GPT-3.5, GPT-4), Google's BERT, and Meta's LLaMA.
- These models have hundreds of billions of parameters and tokens, allowing them to capture intricate patterns in language and generate coherent, contextually relevant responses.

https://en.wikipedia.org/wiki/Large_language_model

Recent Advances and Applications

- LLMs have continued to evolve, with models like OpenAI's Codex (based on GPT-3) fine-tuned for specific tasks such as code generation.
- These models are now used in various applications, from chatbots and virtual assistants to automated content creation and programming assistance.
- LLMs have come a long way from their early days, and they continue to push the boundaries of what AI can achieve in understanding and generating human language.

Book: Foundations of Data Science

- This book thoroughly introduces the fundamental concepts of data science, including probability, statistical inference, linear regression, and machine learning.
- It strongly emphasizes the mathematical and algorithmic foundations of data science, making it particularly valuable for readers who want a deep understanding of the theoretical aspects.

- https://web.eecs.umich.edu/~movaghar/book Machine Learning 2018.pdf

Excerpt from the Book

“ …we have written this book to cover the theory we expect to be useful in the next 40 years, just as an understanding of automata theory, algorithms, and related topics gave students an advantage in the last 40 years. One of the major changes is an increase in emphasis on probability, statistics, and numerical methods…”

Interdisciplinary Nature of Data Science

- The book integrates concepts from statistics, operations research, and computer science, reflecting the interdisciplinary nature of data science.
- It delves into the complexities of high-dimensional data, which is crucial for understanding modern data analysis.
- It covers practical techniques such as singular value decomposition (SVD), random walks, and Markov chains, which are essential for real-world data science applications.

Theoretical Foundations of Machine Learning, Integration, and Ethics

- The book discusses various machine learning algorithms and their theoretical foundations, helping readers understand the principles behind these powerful tools.
- It emphasizes the importance of collaboration between data scientists and domain experts, ensuring that assumptions are balanced with computational efficiency.
- The book also touches on the ethical use of data science, which is increasingly important in today's data-driven world.

Overview

- Background
- What is Artificial Intelligence (AI) for Software Engineering (SE)?
- What are the applications of AI in SE?
  - Can LLMs help us do program verification?
  - Can LLMs help us write unit tests?
  - Can LLMs help us do mutation testing?
  - Can LLMs help us do vulnerability analysis?
  - Can LLMs help us fix bugs and write new code?
  - Can LLMs help us do test generation?

What is Artificial Intelligence (AI) for Software Engineering (SE)?

AI for SE

- Artificial Intelligence for Software Engineering (AI for SE) involves using artificial intelligence and machine learning techniques to enhance and automate various software development and maintenance aspects.
- AI for SE aims to make software development more efficient, reliable, and scalable by leveraging the power of AI.

Automating Software Development

- AI has made tremendous progress in automating numerous jobs typically undertaken by software programmers.
- AI-powered systems, for example, may produce code that fulfills a set of requirements. This method is known as automated programming, and it is becoming more popular.

Improving Software Testing

- AI is altering the way software is tested.
- Algorithms based on artificial intelligence may be used to automate testing, discover and diagnose mistakes, and optimize testing situations.
- This strategy can greatly enhance software quality while lowering testing time and expense.

Improving Software Upkeep

- AI may also aid with software maintenance.
- AI systems can analyze massive volumes of software-related data and make recommendations for upgrades and enhancements using machine learning.
- This method can assist software developers in keeping software systems up to date and improving their overall quality.

Intelligent System Enabling

- AI is also allowing for the creation of intelligent software systems.
- These systems are capable of learning from data and adapting to changing conditions.
- AI-powered chatbots, for example, may learn from prior discussions and improve their replies over time.
- Similarly, recommendation systems can improve their recommendations by learning from user behavior.

Increasing Software Security

- AI can also help to improve software security.
- For example, AI algorithms may discover security flaws in software systems and offer fixes.
- They can also recognize possible risks and take preventative steps.

Addressing the Talent Shortage

- Finally, AI can assist in overcoming the software engineering skills problem.
- AI-powered tools and systems may help software developers be more productive, efficient, and effective.
- This can assist organizations in meeting their software development objectives while using fewer resources.

Code Generation and Debugging

- AI can assist in writing code, reducing the time and effort required by developers.
- AI can identify bugs in the code and even suggest or implement fixes.

Predictive Analytics and Automated Testing

- AI can predict potential issues in software projects, such as delays or resources.
- AI can create and run tests to ensure software quality and reliability.

LLMs for SE

- Many AI applications in software engineering (SE) leverage Large Language Models (LLMs).
- These models have shown significant promise in various aspects of software development.

Code Completion and Code Generation

- Tools like GitHub Copilot use LLMs to provide intelligent code suggestions and auto-completion, enhancing developer productivity.
- LLMs can generate code snippets or even entire functions based on natural language descriptions, making it easier to implement features quickly.

Testing, Debugging and Code Refactoring

- LLMs assist in identifying and fixing bugs by analyzing code and suggesting potential fixes.
- They help improve the structure of existing code without changing its functionality, making the codebase cleaner and more maintainable.
- LLMs can generate test cases based on the code, ensuring better coverage and reliability.

Overview

- Background
- What is Artificial Intelligence (AI) for Software Engineering (SE)?
- What are the applications of AI in SE?
  - Can LLMs help us do program verification?
  - Can LLMs help us write unit tests?
  - Can LLMs help us do mutation testing?
  - Can LLMs help us do vulnerability analysis?
  - Can LLMs help us fix bugs and write new code?
  - Can LLMs help us do test generation?

What are the applications of AI in SE?

Unleashing the Potential of OpenAI's Codex in Software Engineering

- OpenAI's Codex is extensively used in various applications within software engineering (SE).
- Codex, which powers tools like GitHub Copilot, has become a valuable asset for developers by automating repetitive coding tasks, generating code snippets, and even assisting with code completion and debugging.

- https://www.toolify.ai/ai-news/unleashing-the-potential-of-openais-codex-in-software-engineering-2674967
- https://openai.com/index/openai-codex/

Codex

- Codex is an advanced AI model developed by Open AI that translates natural language into code. It is a descendant of OpenAI's GPT-3 model, fine-tuned specifically for programming tasks.
- Based on GPT-3, a neural network trained on text, Codex was additionally trained on 159 gigabytes of Python code from 54 million GitHub repositories.
- Open AI claims that Codex can create code in over a dozen programming languages, including Go, JavaScript, Perle, PHP, Ruby, Shell, Swift, and TypeScript, though it is most effective in Python.

Codex Highlights

- OpenAI's Codex combines natural language understanding with code generation, revolutionizing software development.
- Codex excels at generating code snippets, automating repetitive coding tasks, and assisting beginners in learning programming.
- It has limitations in reasoning abstractly, handling novel or niche concepts, and generating code that meets complex requirements.
- Software engineers can use Codex as a tool to augment their work, ensuring human intervention to produce reliable code.
- The future possibilities of Codex include automated bug detection, code refactoring, and intelligent code completion.

Can LLMs help us do program verification?

Result 1: LLM Capabilities in Loop Invariant Synthesis

- The authors observe that Large Language Models (LLMs) like GPT-3.5 and GPT-4 can synthesize loop invariants for a class of programs in a zero-shot setting.
- However, they require multiple samples to generate the correct invariants.

https://web.eecs.umich.edu/~movaghar/Sarah Fakhoury 2024.pdf

Leveraging LLMs

- The approach leverages an LLM for generation and ranks using a purely neural model and does not require a program verifier at the inference time.
- This approach involves designing a ranker that can distinguish between correct and incorrect invariants based on problem definition.
- The ranker is optimized as a contrastive ranker, which helps in prioritizing the most promising invariants for verification.

Ranking Mechanism

- The paper introduces a ranking mechanism to evaluate and prioritize the generated loop invariants.
- This helps in reducing the number of calls to a program verifier, making the process more efficient.

Empirical Evaluation

- The authors conduct an empirical evaluation to demonstrate the effectiveness of their approach.
- They show that their ranking mechanism significantly improves the performance of LLMs in generating correct loop invariant.
- These contributions aim to enhance the usability and efficiency of LLMs in program verification tasks, particularly in the context of loop invariant synthesis.

Result 2: LEMUR: INTEGRATING LARGE LANGUAGE MODELS IN AUTOMATED PROGRAM VERIFICATION

- The paper introduces a novel framework that integrates LLMs with automated reasoners for program verification.
- This framework leverages LLMs' high-level reasoning capabilities and automated reasoners' precise low-level reasoning.
- The authors present LEMUR as a proof system and provide formal proof of its soundness.
- This is the first formalization of such a hybrid approach, demonstrating that the integration of LLMs and automated reasoners can be both sound and effective.
- https://web.eecs.umich.edu/~movaghar/LEMUR 2024.pdf

Sound and Terminating Algorithm

- The paper describes an instantiation of the LEMUR calculus that results in a sound and terminating algorithm.
- This ensures that the verification process is reliable and can be completed within a finite amount of time.

Implementation and Optimizations, Evaluation and Results

- The authors implement the proposed framework and introduce several practical optimizations to enhance its performance.
- These optimizations make the framework more efficient and applicable to real-world verification tasks.
- The paper includes an evaluation of the framework, demonstrating its effectiveness in various program verification scenarios.
- The results show that the integration of LLMs and automated reasoners can significantly improve the verification process.

Can LLMs help us write unit tests?

Result 3: Unit Test Generation

- LLMs can generate unit tests by analyzing the code and creating test cases that cover various scenarios.
- The generated tests can achieve higher code coverage and better quality by using LLMs.
- These models can identify edge cases and generate tests that human developers might overlook
- https://web.eecs.umich.edu/~movaghar/Multi-language Unit Testing LLM 2024.pdf

Result 4: Natural Language Processing

- LLMs can understand and generate human-like text, making the tests readable and maintainable.
- This helps in creating tests that are closer to what a developer might write.
- https://web.eecs.umich.edu/~movaghar/Unit Test Generation LLM 2024.pdf

Result 5: Empirical Study of Unit Test Generation with LLMs

This study investigates the effectiveness of using LLMs for generating unit tests compared to traditional tools like EvoSuite.

It evaluates various open-source LLMs and their performance in generating unit tests for Java projects.

The findings highlight the potential of LLMs in this domain while also identifying areas for improvement.

https://web.eecs.umich.edu/~movaghar/EVosuite-LLM-2024-1.pdf

Result 6: Large-scale Study on LLMs for Test Case Generation

- This comprehensive study assesses the capabilities of several LLMs, including GPT and Mistral, for generating unit tests.
- The research compares the correctness, understandability, coverage, and bug-detection capabilities of LLM-generated tests against those produced by EvoSuite.
- The results indicate that while LLMs show promise, there are still challenges to be addressed to match the effectiveness of traditional methods.

- https://web.eecs.umich.edu/~movaghar/Evosuite-LLM-2024-2.pdf

Result 7: Meta's TestGen-LLM

- Meta's TestGen-LLM tackles the time-consuming task of unit test writing by leveraging the power of Large Language Models (LLMs).
- General-purpose LLMs like Gemini or ChatGPT might struggle with the specific domain of unit test code, testing syntax, and generating tests that don't add value.
- TestGen-LLM is specifically tailored for unit testing.

https://www.freecodecamp.org/news/automated-unit-testing-with-testgen-llm-and-cover-agent/

Can LLMs help us do mutation testing?

Result 8: An Exploratory Study on Using Large Language Models for Mutation Testing

- This paper investigates the performance of LLMs in generating effective mutations, focusing on their usability, fault detection potential, and relationship with real bugs.
- https://web.eecs.umich.edu/~movaghar/Mutation-Testing-LLMS-2024.pdf

Result 9: Mutation-based Consistency Testing for Evaluating the Code Understanding Capability of LLMs

- This study introduces a method to assess the code understanding performance of LLMs by applying code mutations to existing code generation datasets.

- https://web.eecs.umich.edu/~movaghar/Mutattion-Testing-Code-Understanding-LLMs-2024.pdf

Result 10: A Mutation Testing Framework of In-Context Learning Systems

- This paper proposes a mutation testing framework specifically designed for in-context learning systems, leveraging LLMs to evaluate the quality and effectiveness of test data.
- https://web.eecs.umich.edu/~movaghar/Mutation-Testing-Framework-LLMs-2024 .pdf

Result 11: On the Use of Large Language Models for Mutation Testing

- The authors conducted a large-scale empirical study involving six large language models (LLMs) and 851 real bugs from two Java benchmarks (Defects4J 2.0 and ConDefects) to evaluate the effectiveness of LLMs in generating mutations.
- The study found that LLMs generate more diverse mutations that are behaviorally closer to real bugs, leading to approximately 19% higher fault detection compared to existing approaches.

- https://web.eecs.umich.edu/~movaghar/Mutation Testing LLM 2025.pdf

Challenges and Prompt Engineering

- Despite their effectiveness, the mutants generated by LLMs had lower compilability rates and higher useless and equivalent mutation rates compared to rule-based approaches
- The paper also explores alternative prompt engineering strategies and identifies the root causes of uncompilable mutations, providing insights for improving the performance of LLMs in mutation testing.

Table 1. Real Bugs Used in Our Experiment

Dataset	Project	# of Bugs	Time Span
Defects4J (D4J)	Math	106	2006/06/05 - 2013/08/31
	Lang	65	2006/07/16 - 2013/07/07
	Chart	26	2007/07/06 - 2010/02/09
	Time	27	2010/10/27 - 2013/12/02
	Closure	133	2009/11/12 - 2013/10/23
	Mockito	38	2009/06/20 - 2015/05/20
	Cli	39	2007/05/15 - 2018/02/26
	Codec	18	2008/04/27 - 2017/03/26
	Csv	16	2012/03/27 - 2018/05/18
	Gson	18	2010/11/02 - 2017/09/21
	JacksonCore	26	2013/08/28 - 2019/04/05
	Jsoup	93	2011/07/02 - 2019/07/04
ConDefects (CD)	—	246	2024/03/01 - 2024/06/30
Total	—	851	2006/07/16 - 2024/06/30

Table 2. Studied LLMs

Model Type	Studied Model	Base Model	Training Data Time	Release Time	Size
Closed	**GPT-3.5-Turbo**	GPT	2021/09	2023/03	–
	GPT-4o	GPT	2023/10	2024/05	–
	GPT-4o-Mini	GPT	2023/10	2024/07	–
Open	**StarChat-β-16b**	StarCoder	–	2023/06	16B
	CodeLlama-Instruct-13b	Llama	–	2023/08	13B
	DeepSeek-Coder-V2-236b	DeepSeek	2023/09	2024/07	236B

https://blog.spheron.network/choosing-the-right-llm-2024-comparison-of-open-source-vs-closed-source-llms

Table 3. Default Few-Shot Examples from QuixBugs

Correct Version	**Buggy Version**
`n = (n & (n - 1));`	`n = (n ^ (n - 1));`
`while (!queue.isEmpty())`	`while (true)`
`return depth==0;`	`return true;`

`ArrayList r = new ArrayList();`
`r.add(first).addll(subset);`
`to_add(r);`
	`to_add.addAll(subset);`

`c = bin_op.apply(b,a);`	`c = bin_op.apply(a,b);`
`while(Math.abs(x-approx*approx)>epsilon)`	`while(Math.abs(x-approx)>epsilon)`

'{WHOLE_JAVA_METHOD}'

Context (Whole Method)

Above is the original code. your task is to generate '{MUT_NUMBER}' mutants, (notice: mutant refers to the mutant in software engineering, i.e., making subtle changes to the original code) in: '{CODE_ELEMENT}', as follows are some examples of mutants which you can refer to:

Instructions and ***Input Data***

```
{…
   "precode": "while (Math.abs(x-approx*approx) > epsilon) { "
   "aftercode": " while (Math.abs(x-approx) > epsilon) {",
…}
```

Context (Few-Shot Examples)

#Requirement:

1. Provide generated mutants directly
2. A mutation can only occur on one line
3. Your output must be like:

   [{ "id":, "line":, "precode":"", "filepath":"kk", "aftercode":"" }],

   where "id" stands for the mutant serial number, "Line" represents the line number of the mutated, "precode" represents the line of code before mutation and it can not be empty, "aftercode" represents the line of code after mutation
4. Prohibit generating the exact same mutants
5. All write in a JSON file

Output Indicator

Fig. 1. The Default Prompt Template

Table 4. Overall Performance of All the Mutation Generation Techniques

Metric	GPT-3.5	GPT-4o	GPT-4o-M	DC-236b	SC-16b	CL-13b	LEAM	μBert	PIT	Major
Mut. Score	0.697	0.726	0.708	0.694	0.516	0.700	0.545	0.675	0.581	0.486
Mut. Count	538338	539715	588803	594711	463248	457195	890417	200896	2890433	346282
Avg. Gen. Time	1.79	1.76	1.65	4.25	7.53	9.06	3.06	2.33	0.02	0.08
Comp. Rate	60.2%	75.6%	73.6%	75.5%	11.1%	70.2%	35.0%	22.5%	——	98.6%
Useless Mut. Rate	10.9%	7.8%	6.7%	8.3%	8.9%	39.3%	1.0%	1.7%	0.0%	0.0%
Eq. Mut. Rate	2.2%	1.2%	1.7%	1.2%	1.8%	1.2%	1.3%	2.5%	0.0%	0.6%
Real Bug Detec.	91.7%	93.4%	93.4%	92.8%	47.1%	83.1%	71.7%	71.3%	51.3%	74.4%
Coupling Rate	41.4%	43.6%	44.1%	41.5%	29.4%	39.8%	28.7%	41.6%	14.0%	36.0%
Ochiai Coeff.	65.0%	68.5%	66.9%	61.8%	19.3%	40.8%	37.4%	31.6%	31.5%	44.4%

1. How many LLMs have a Mutation Score of more than 70%?
2. How many LLMs have Real Bug Detection of more than 90%?
3. How many LLMs achieve a Compilability rate of more than 70%?
4. Which LLM excels w.r.t. Compilability Rate and Equivalent Mutation Rate?

Answers

1. 6
2. 4
3. 4
4. Major

Table 12. Performance Under the Same Number of Mutations

Metric	GPT-3.5	GPT-4o	GPT-4o-M	DC-236b	SC-16b	CL-13b	LEAM	μBert	Major	PIT
Comp. Rate	62.9%	77.4%	70.1%	73.0%	39.3%	74.2%	38.9%	18.5%	96.5%	——
Useless Mut. Rate	11.0%	7.3%	7.0%	8.9%	27.1%	39.0%	1.4%	1.4%	0.0%	——
Real Bug Det.	89.2%	90.8%	91.3%	89.7%	58.0%	73.5%	67.5%	55.0%	83.3%	43.7%
Coupling Rate	13.3%	14.6%	12.6%	13.4%	11.9%	10.5%	13.2%	13.2%	12.8%	2.4%
Ochiai Coefficient	35.8%	45.0%	40.8%	35.3%	15.3%	17.8%	17.0%	15.0%	34.1%	19.2%

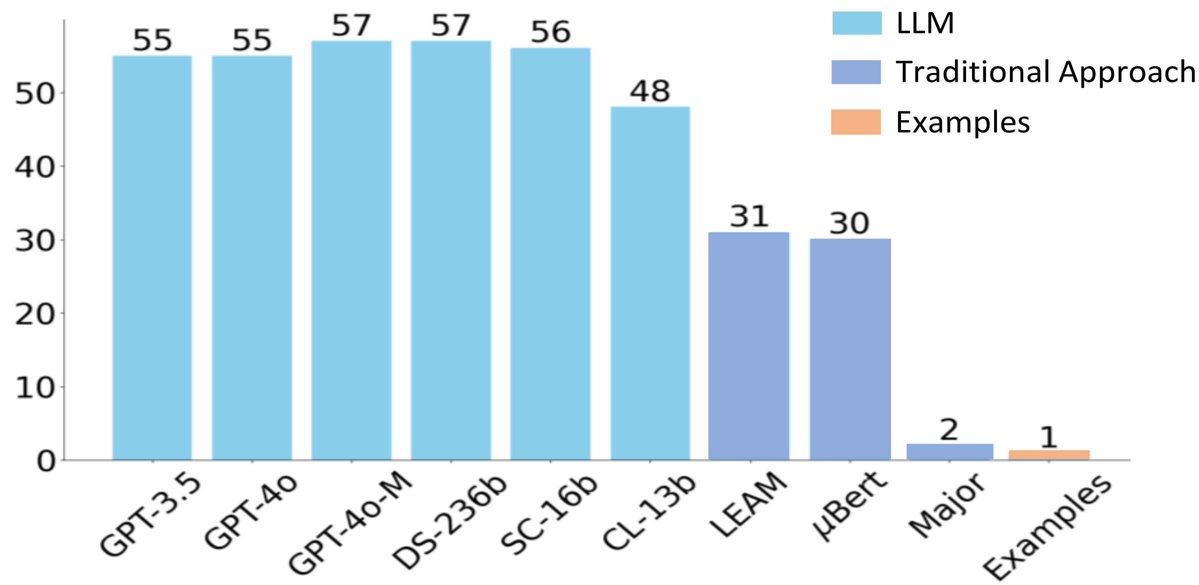

Fig. 2. Number of newly introduced AST nodes of the studied approaches and the few-shot examples.

- Finding 1: Rule-based approaches cost less time than LLMs in generating mutations. For example, GPT LLMs take fewer than 1.8s per mutation. In contrast, PIT and Major only require 0.02s and 0.08s, respectively, while small model-based approaches (i.e., LEAM andμBert) take 2-3s.
- Finding 2: Rule-based approaches, PIT and Major, outperform others in terms of compilability rate, useless mutation rate, and equivalent mutation rate. In particular, GPT-4o exhibits a compilability rate of 75.6%, a useless mutation rate of 7.8%, and an equivalent mutation rate of 1.2%. In contrast, Major excels with a compilability rate of 98.3%, a useless mutation rate of 0%, and an equivalent mutation rate of 0.6%.

- Finding 3: LLM-based mutation approaches introduce more types of new AST nodes than traditional approaches, while less inclined to generation deletion mutations.
- Finding 4: Five out of six LLMs significantly outperform traditional approaches in Real Bug Detectability. In particular, on ConDefects, GPT-4o outperforms nearly 29% over the best of the conventional approach, Major, highlighting the advantages of LLMs in detecting real bugs.
- Finding 5: GPT-4 achieves the highest Coupling Rate at 44.1%, outperforming all traditional approaches, including the best traditional approach, μBert, by 2.5%.
- Finding 6: Five LLMs surpass all traditional generation approaches in the Ochiai Coefficient. In particular, the best LLM, GPT-4o, exceeds the best of traditional approaches, Major, by 24.1%.

- Finding 7: Two GPT-4 models perform best, while DeepSeek has a similar performance to GPT -3.5, standing out among open-source models. The newer models exhibit better performance in mutation generation.
- Finding 8: The prompt with the whole method and few-shot examples as context (i.e., P1) achieves the best performance across all Behavior Metrics, whereas adding the code of test suites (i.e., P4) decreases performance.
- Finding 9: The mutations generated by GPT models have nine main types of compilation errors, with Usage of Unknown Methods and Code Structural Destruction being the two most prevalent types.
- Finding 10: Member references and method invocations within the code context are the most likely triggers for LLMs to generate non-compilable mutations.

Can LLMs help us do vulnerability analysis?

Result 12: LLMs in Source Code Vulnerability Detection

- This paper discusses how LLMs can be used to analyze source code and detect known vulnerabilities.
- It highlights the use of LLMs to capture complex patterns in code and convert source code to intermediate representations for better analysis.
- https://web.eecs.umich.edu/~movaghar/Vulnerability-detection-LLMs-2024.pdf

Result 13: LLM-based Agents for Software Engineering

- This survey paper reviews the current state of LLM applications in software engineering, including vulnerability and defect detection.
- It covers various topics, such as code generation, autonomous decision-making, and software maintenance.
- https://web.eecs.umich.edu/~movaghar/LLM-based-agent-se-2024.pdf

Result 14: Large Language Model for Vulnerability Detection and Repair

- This systematic literature review examines approaches aimed at improving vulnerability detection and repair through LLMs.
- It covers research from leading software engineering, AI, and security conferences and journals.
- https://web.eecs.umich.edu/~movaghar/Source-code-valnurability-detection-LLMs-2024.pdf

Can LLMs help us fix bugs and write new code?

Result 15: Teaching Large Language Models to Self-Debug

- The paper introduces a self-debugging approach that enables Large Language Models (LLMs) to identify and correct their mistakes without human feedback. This is achieved through few-shot demonstrations.
- It implements a method where the LLM performs "rubber duck debugging," explaining its generated code in natural language to identify errors by investigating execution results.

https://web.eecs.umich.edu/~movaghar/Self-Debgging -LLM 2023.pdf

Accuracy and Efficiency Improvement

- It demonstrates that the self-debugging approach achieves state-of-the-art performance on several code generation benchmarks, including Spider (text-to-SQL), TransCoder (C++-to-Python translation), and MBPP (text-to-Python generation).
- It shows significant improvements in prediction accuracy, particularly on complex problems. For example, it improves baseline accuracy by up to 12% on benchmarks with unit tests.
- It highlights that leveraging feedback messages and reusing failed predictions notably improves sample efficiency, matching or outperforming baseline models that generate more than 10 times the number of candidate programs.

Result 16: Evaluating LLMs at Detecting Errors in LLM Responses

- The paper introduces ReaLMistake, the first error detection benchmark that consists of objective, realistic, and diverse errors made by LLMs.
- This benchmark includes three challenging tasks that introduce objectively assessable errors in four categories: reasoning correctness, instruction-following, context-faithfulness, and parameterized knowledge.

https://web.eecs.umich.edu/~movaghar/LLM Error Detection 2024.pdf

Evaluation of Error Detectors and the Analysis of Explanations

- The authors use ReaLMistake to evaluate error detectors based on 12 different LLMs.
- They find that top LLMs like GPT-4 and Claude 3 detect errors at very low recall rates, and all LLM-based error detectors perform significantly worse than humans.
- The paper highlights that explanations provided by LLM-based error detectors lack reliability.
- This finding underscores the need for more robust methods to explain and justify the detected errors.

Sensitivity to Prompt Changes and Evaluation of Improvement Approaches

- The study shows that LLM-based error detection is highly sensitive to small changes in prompts, making it challenging to improve the performance of these detectors.
- The paper evaluates popular approaches to improving LLMs, such as self-consistency and majority vote, and finds that these methods do not enhance error detection performance.

Result 17: Enhancing the Code Debugging Ability of LLMs

- This paper introduces DEBUGEVAL, a benchmark designed to evaluate the debugging capabilities of LLMs.
- It proposes a framework called MASTER to enhance debugging abilities through data refinement and supervised fine-tuning.
- https://web.eecs.umich.edu/~movaghar/Code-Debugging-LLMs-2024.pdf

Result 18: LLMs for Software Engineering

- This comprehensive review covers various applications of LLMs in software engineering, including debugging automation.
- It analyzes methods used in data collection, preprocessing, and application, highlighting the role of well-curated datasets.
- https://web.eecs.umich.edu/~movaghar/LLM-SE-Review-2024.pdf

Result 19: LLM Assisted Software Engineering

- This paper provides an overview of the current state-of-the-art in LLM support for software construction, including debugging.
- It illustrates the potential and challenges of using LLMs in software engineering tasks.
- https://web.eecs.umich.edu/~movaghar/LLM-Assisted-SE-2023-Review.pdf

Can LLMs help us do test generation?

Result 20: CoverUp: Coverage-Guided LLM-Based Test Generation

- The paper introduces CoverUp, a system that combines coverage analysis with Large Language Models (LLMs) to generate high-coverage Python regression tests.
- It utilizes an iterative process where coverage information is used to guide the LLM in refining tests to cover more lines and branches of code.

https://web.eecs.umich.edu/~movaghar/Coverup Regression Testing 2024.pdf

Coverage Improvement

- The paper demonstrates through empirical analysis that CoverUp significantly improves test coverage compared to existing methods.
- For example, it achieves a median line+branch coverage of 80% per module, compared to 47% by CodaMosa, and an overall coverage of 90%, compared to 77% by MuTAP.
- The paper highlights that the iterative, coverage-guided approach is crucial to its success, contributing to nearly 40% of its effectiveness.

Result 21: Automated Unit Test Improvement using Large Language Models at Meta

- The paper introduces TestGen-LLM, a tool that uses Large Language Models (LLMs) to automatically improve existing human-written unit tests.
- It demonstrates that TestGen-LLM can generate additional test cases that cover previously missed corner cases, thereby increasing overall test coverage.
- It implements a set of filters to ensure that the generated test classes provide measurable improvements over the original test suite, reducing issues related to LLM hallucination.

https://web.eecs.umich.edu/~movaghar/Automatic Test Generation Meta 2024.pdf

Increased Reliability and Coverage of Test Cases

- The paper describes the deployment of TestGen-LLM at Meta's test-a-thons for Instagram and Facebook platforms, where it improved 11.5% of all classes to which it was applied.
- It reports that 75% of TestGen-LLM's test cases were built correctly, 57% passed reliably, and 25% increased coverage.
- Additionally, 73% of its recommendations were accepted for production deployment by Meta software engineers.

Result 22: Large Language Models as Test Case Generators: Performance Evaluation and Enhancement

- The paper conducts an extensive evaluation of Large Language Models (LLMs) in generating test cases.
- The study finds that the performance of LLMs declines significantly when handling more complex problems, often resulting in errors in the generated test cases.

- https://web.eecs.umich.edu/~movaghar/LLM Test Case Generators 2024.pdf

Improved Accuracy of Test cases

- It proposes a multi-agent framework called TestChain, which decouples the generation of test inputs and test outputs. This framework uses a ReAct format conversation chain for LLMs to interact with a Python interpreter, leading to more accurate test outputs.
- It demonstrates that TestChain significantly outperforms the baseline. Specifically, using GPT-4 as the backbone, TestChain achieves a 13.84% improvement in the accuracy of test cases on the LeetCode-hard dataset.

Software Testing with Large Language Models: Survey, Landscape, and Vision

- The paper provides a comprehensive review of the utilization of large language models (LLMs) in software testing.
- It analyzes 102 relevant studies, highlighting the various software testing tasks for which LLMs are commonly used, such as test case preparation and program repair.
- The paper discusses the types of LLMs employed, the prompt engineering techniques used, and the accompanying methods that enhance their effectiveness.
- https://web.eecs.umich.edu/~movaghar/Testing LLMs Survey 2024.pdf

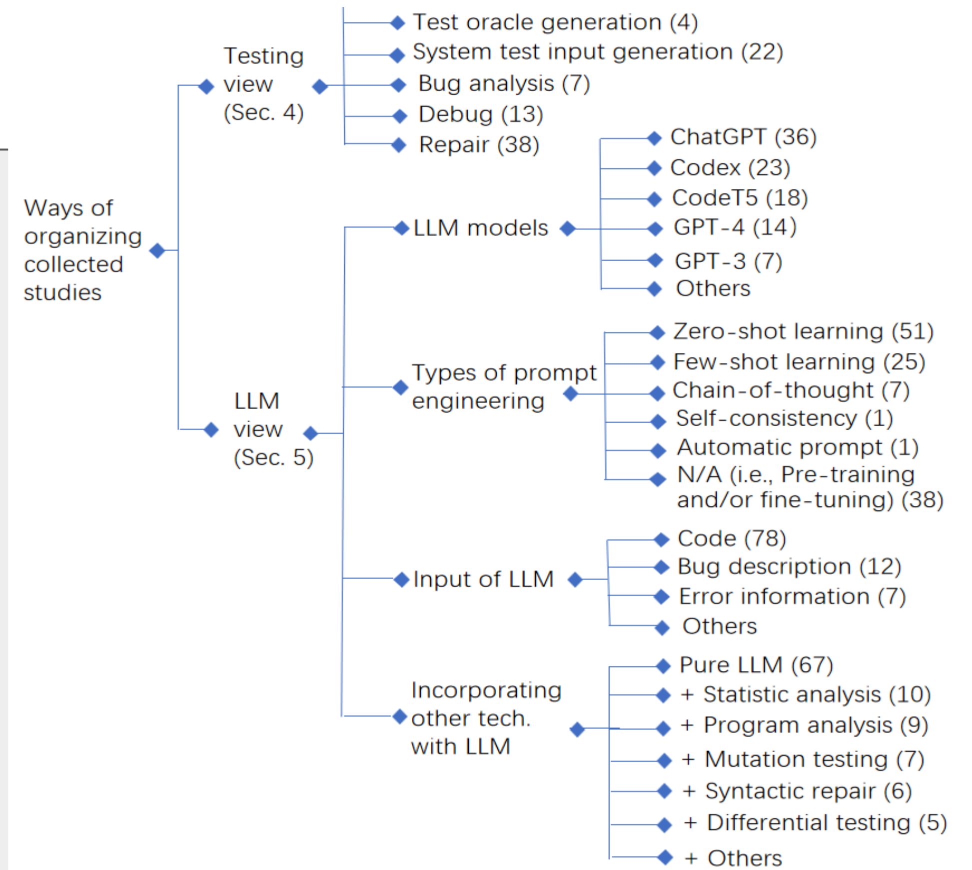

Figure 1: Structure of the contents in this paper (the numbers in bracket indicates the number of involved papers, and a paper might involve zero or multiple items)

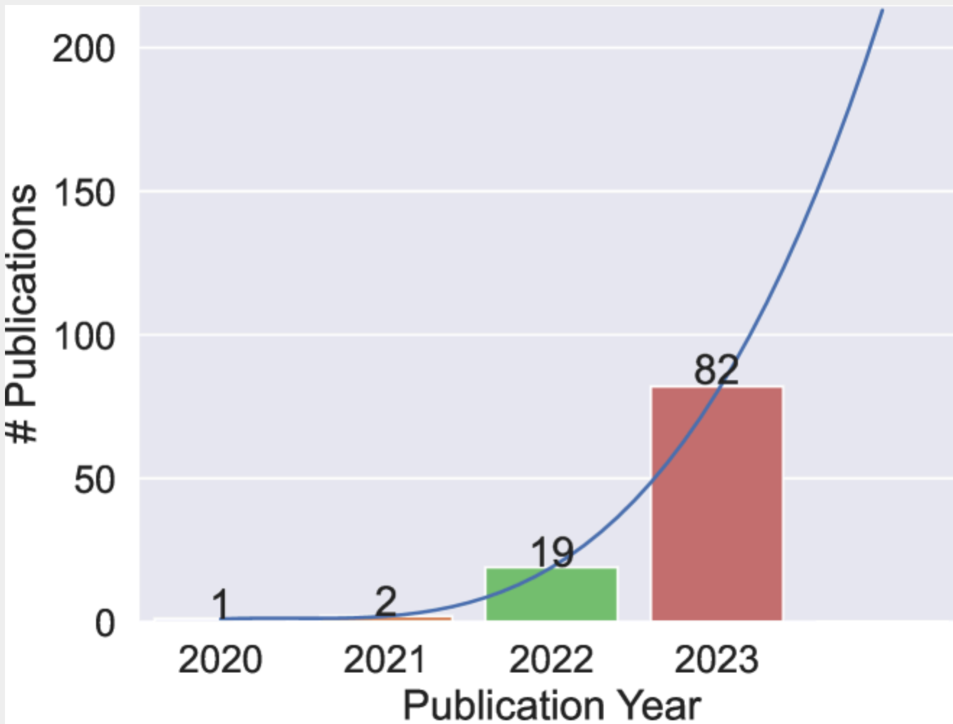

Figure 3: Trend in the number of papers with year

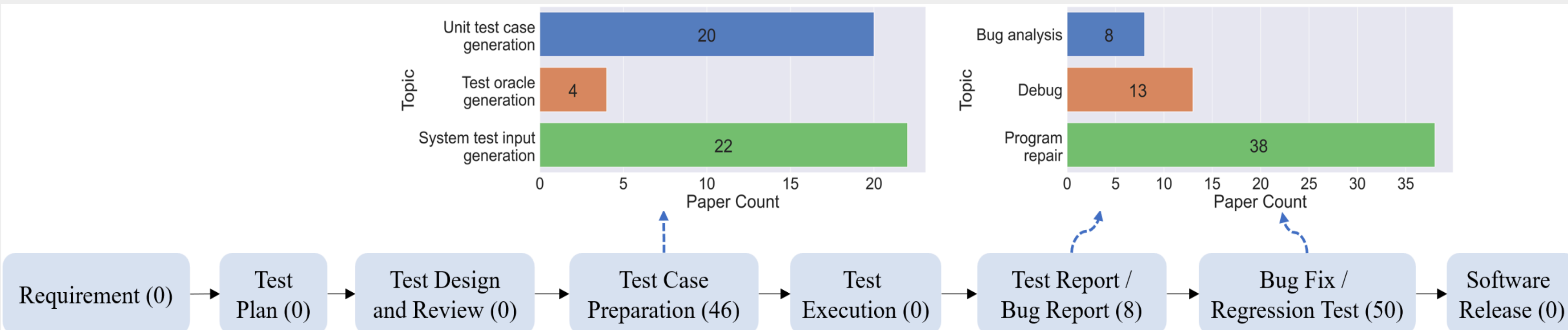

Figure 4: Distribution of testing tasks with LLMs (aligned with software testing life cycle [134, 135, 136], the number in bracket indicates the number of collected studies per task, and one paper might involve multiple tasks)

TABLE III: Performance of unit test case generation

Dataset	**Correctness**	Coverage	LLM	Paper
5 Java projects from Defects4J	16.21%	5%-13% (line coverage)	BART	[26]
10 Jave projects	40%	89% (line coverage), 90% (branch coverage)	ChatGPT	[36]
CodeSearchNet	41%	N/A	ChatGPT	[7]
HumanEval	78%	87% (line coverage), 92% (branch coverage)	Codex	[39]
SF110	2%	2% (line coverage), 1% (branch coverage)	Codex	[39]

Note that, [39] experiments with Codex, CodeGen, and ChatGPT, and the best performance was achieved by Codex.

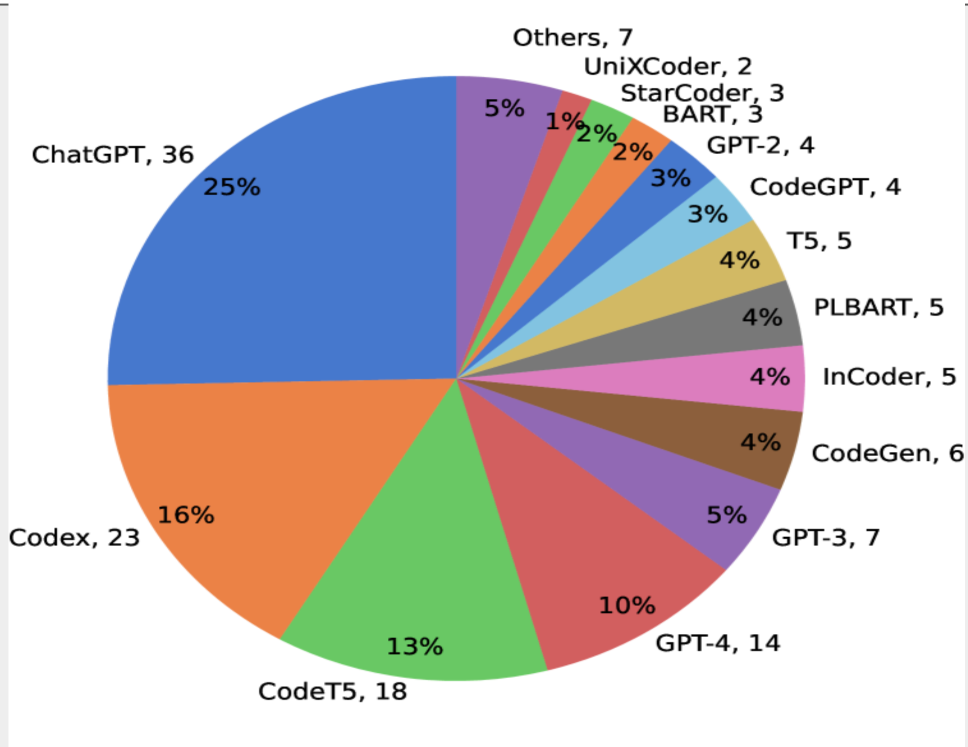

Figure 6: LLMs used in the collected papers