Group Assignment Brief
CO7095 Software Measurement and Quality Assurance 2024-25 SEM1

0. AI Policy

You should not use any AI tools to create any of the gradable content/component in this group assignment. However, you may use AI tools to help you check for grammatical and spelling errors. If we detected any use of AI tools in your group assignment, other than for checking of grammatical and spelling errors, it will be considered as a case of plagiarism, and you will be referred to the plagiarism officer. Additionally, you will get a zero mark in this group assignment immediately.

1. Overview of Group Assignment

The assignment requires students to work in teams to develop a fully functional software product/application, focused on the backend functionality and its overall software quality, based on the quality assurance techniques taught in this module. There is no need for a GUI frontend. A simple text-based frontend is sufficient. There is also no need for any databases. You can use a serialized text file for storage. Using a GUI frontend and a database will not give you additional marks, and if you implement those, it may cause you to slip the project since these consume time to implement, and you may not have sufficient time for other required features (*check Marking Rubric below*). The development process should follow the agile software development methodology and other software quality methods taught in lectures. There is also a research component (*on the topic of Automatic Code Generation*). Automatic Code Generation may improve the quality and speed of development, but it is still not very matured. You will need to provide critical analysis on a research paper on this topic, from the list of research papers in Section 9. You are not allowed to change this topic, nor choose a different paper that is not from Section 9.

Please see below for the detailed requirements on the programming language, research component, development and planning tools to be used.

Programming Language	Only Java (*You must not use other languages as we are using some of the software quality tools developed in Java and you are required to use them in your group assignment, and this is also for marking standardization.*)
Research Component	Choose one research paper from Section 9 and provide critical analysis on it during the reading week. Present the summary of the paper and its limitations. Propose potential solutions to solve the existing limitations found in the paper.
Development & Planning tools	We will be using Eclipse, GitHub, and other tools introduced in the module. For development tools, you must use **Eclipse** since all teaching staffs are familiar with this tool and you can use some of useful plug-ins to evaluate the proposed test cases. We will load your Eclipse project in one of the lab computers, and if it not loadable, you will not get any marks! You must use GitHub for project management (*through GitHub Project*), code commits and code review. You

	must also add the GitHub ID of the Professor and all the TAs into your group assignment repository.

Each group must choose one (***no two groups and beyond are allowed to choose the same project***) project from the provided **CO7095 project list (*Section 8*)** and propose **fixed-sized** user stories (*for a group of 5 members, this size should be at least 10 for each member*) that are to be developed for the chosen project. You must explicitly state which member of the team is doing which user stories in the report. For the research component, each group will choose one research paper (*no two groups and beyond are **allowed** to choose the same research paper*) from the provided list of **research papers (***Section 9*). The choosing of the project topic and the research paper from Section 8 and 9 is first come first serve. The professor will share a document to all groups to indicate their chosen project topic and research paper.

Each user story should be done through appropriate market research and team discussion. You should bear in mind that this group project only last approximately one month – You should not do something that is very complicated and that it will take more than one month. You should also start early and not wait until the last minute. **Note that** each group should first build the **backbone** of the project by defining the corresponding empty classes and methods based on the user stories. Within each sprint, each group member needs to implement the assigned empty classes or methods. Every user story (*issue*) in the GitHub Project needs to have a GitHub branch linked to it (*even though you may later merge this branch into the main branch, it should not be deleted to provide evidence of development*).

From the module learning outcomes, this assignment assesses:

1. Participation in software quality strategies that are taught in lectures, for the effective working of an agile team towards a common goal.
2. Implementation of agile methodologies in a group-based setting while engaging in a practical software project to produce a functional quality software product.
    a. There should be **at least 2 sprints (*each sprint is minimum of 2 weeks*)**.
    b. Each team member in the group should be assigned an **equal** number of **user stories**, in which each user story should be estimated with the appropriate **story points** via **Planning Poker**. The total story points for each team member in the group should also be relatively equal.
    c. **PERT** should be used to measure the **critical path**.
    d. Estimate the cost using **COCOMO.**
3. Implementing test cases using the taught testing tools and techniques for **Black-Box testing** strategies/algorithms (***Specification-Based and Random-Based***) and **the White-Box testing** strategies/algorithms (***Statement-Based and Branch-Based*)**
4. Measuring the **test coverage** to check how much code has been tested. For each testing technique, you should create a separate test package in your repository that will include **all the relevant test cases** for black-box "*test.blackbox*" and white-box "*test.whitebox*", so that Eclipse can generate the corresponding test coverage results for each strategy (Black-Box and White-Box).
5. Provide critical analysis on a research paper about **Automatic Code Generation**. Include its summary and limitation and suggest improvement for the limitation (*choose and read one of the given research papers in Section 9 – provide a summary of the paper, as well as its limitation and suggest how to improve on it*). Each research paper from Section 9 can only be chosen by one group.

Deliverables to be submitted:

1. A video that demonstrates your final deliverable software product and introduces your development progress. It should be a formal presentation, and every member in your team should participate (**at most 5 minutes for each team member**). Each member can discuss within the team to decide who should present what points (see below) and all the points should be covered. The video demonstration should:
    - Introduce the role of each team member.
    - Introduce the motivation and design of your software/application.
    - Demonstrate the features of your software product.
    - Show the critical path and its calculation.
    - Present the estimated story points via planning poker with evidence.
    - Present the cost computed via COCOMO and its calculation.
    - For each sprint, show the following details in your SCRUM (GitHub Project) board:
        - The user stories in each sprint.
        - Who is responsible for each user story.
        - The start and end date of each user story.
    - Show the GitHub commit history of your project that is linked to the user stories.
    - Show your Black-Box test history: What testing tools and techniques did you use? All the test cases and the test results (whether it passed or failed). If any of the test cases failed, you must provide an explanation – a failed test case means your code has bugs and they are not fixed! Ideally, you should not have any failing test cases.
    - Show your White-Box test history: What testing tools and techniques did you use? All the test cases and the test results (whether it passed or failed). If any of the test cases failed, you must provide an explanation – a failed test case means your code has bugs and they are not fixed! Ideally, you should not have any failing test cases.
    - Show your test coverage of your code. For each testing technique, you should create a related test package, which includes **all the relevant test cases** that can be run without errors when Eclipse is loaded to generate the corresponding test coverage result.
    - Show the research on Automatic Code Summarization.

2. A report (**no more than 3000 words**) with two main sections: Section A introduces the software/application features and development techniques (*you should emphasize what software qualities features or techniques that were taught in lectures are used here*), and Section B describes the development process (*you should emphasize the team structure and contribution*). The report structure (you must adhere to the report structure) is shown below:

Section A

- Introduction
- Software planning & design
    - Includes user stories, story points, PERT diagram, critical path, estimated costs via COCOMO, etc.

- Implementation
  - Back-End
- Testing and Test Coverage Measurement
  - Black-Box Testing: What testing tool did you use? All the test cases and the test results (whether it passed or failed). If any test cases failed, you must give an explanation – a failed test case means your code has bugs and they are not fixed!
  - White-Box Testing: What testing tool did you use? All the test cases and the test results (whether it passed or failed). If any test cases failed, you must give an explanation – a failed test case means your code has bugs and they are not fixed!
  - Test Coverage of the individual testing technique.
- Automatic Code Generation (*The below 3 points should be around 800 words*)
  - Summary of a research paper on Automatic Code Generation (*from the list of research papers in Section 9*).
  - Limitation of the chosen research paper.
  - Improvement/Suggestions to the chosen research paper.
- Conclusion

Section B

- Introduction
- Team Structure
- Team Contribution & Reflection

Reference

Appendices

---

3. The source code (with all the test cases) and other relevant artifacts of your software product in a .zip file. Include a detailed README file on how to uncompress, import from the computer in Percy Gee laboratory, compile and run all the code and test cases.

Important Dates:

- Handed out: **16-Oct-2024 at 4:00 pm UK Time**
- Deadline: **17-Jan-2025 by mid-day 12 pm UK Time** (*You must submit the zip file to the provided Turnitin Link in the CO7095 Sem 24-25 Blackboard: "Assessment and Feedback" →"CO7095 Group Assignment". Only 1 member of the group needs to submit. Do not submit the same copy multiple times by different group members. For every group, we will mark only the most recent uploaded copy, based on the uploaded time.*)

2. Other Important Information

- This assignment counts to **70%** of the module's mark.
- This assignment is a group work.
- Please remember to limit your submission file size to under 50MB and submit it before the deadline (any late submission will receive penalty in marks according to the university policy).

- If your video is larger than 50MB, please use the 'video compress' tool to compress it, which you can find in the Resource section.
    - Refer to this student guide on how to upload your video.
- Your submission should include:
    - A report in a PDF file, containing the following:
        - Academic integrity signature signed by all team members.
        - The main text of the report **(no more than 3000 words).** Detailed guidelines of the report can be found under the "Deliverables to be submitted" section.
    - The source code (with all the test cases) and other relevant artifacts of your software product. Detailed guidelines of the report can be found under the "Deliverables to be submitted" section.
    - A video demonstrating your software/application and development process. Detailed guidelines of the video demonstrations can be found under the "Deliverables to be submitted" section.
- You can re-submit your work **multiple times before the deadline**. However, we will only grade the latest uploaded submission. If you have submitted after the deadline (*even if you have a previous submission before the deadline*), we will mark the latest uploaded submission, which is the latest submission after the deadline, and late penalty will be imposed!
- You could submit your report in your first attempt to check for similarity. Then, you can submit your final zip file in your future attempts.
- The similarity score of your submission should be **less than approximately 20%**. You should aim to keep the similarity score to less than 10%.
- The deadline is strict. Penalties in marks will be applied to late submissions according to the university policy.

3. Resit

- We **strongly** advise you to pass the exam via the normal group assignment route.
- If you fail, you will have to accomplish a harder resit exam **individually** at a much later resit period, which may delay your graduation date.
- In the resit exam, you need to answer some short answer questions, some long answer questions as well as some application-based questions.

4. Marking Rubric

Category & Expectation	Outstanding (85 - 100%)	Excellent (70 - 84%)	Competent (60 - 69%)	Satisfactory (50 - 59%)	Unsatisfactory (0 – 49%)
Sprint Planning (10 marks) & Expect to successfully design and analyse an agile project.	Presents more than two completed sprints (*evidenced in the group GitHub project*). Each user story has the story points estimated via Planning Poker with evidence.	Presents two completed sprints (*evidenced in the group GitHub project*). Each user story has the story points estimated via Planning Poker with evidence.	Presents two completed sprints (*evidenced in the group GitHub project*). Some user stories do not have story points estimated via Planning Poker with evidence.	Presents one completed sprint (*evidenced in the group GitHub project*). Each user story has the story points estimated via Planning Poker with evidence.	Presents one or no completed sprint (*evidenced in the group GitHub project*). The user stories do not have story points estimated via Planning Poker with evidence or there is little evidence of planning.
Software Development Measurement (20 marks) & Expect to understand and apply the measurement techniques.	Critical path is analysed using PERT and it is constructed correctly with Network Diagram, calculation and with very detailed explanation (*how it is estimated and calculated*) on all the components in the Network Diagram. The cost is estimated via COCOMO correctly.	Critical path is analysed using PERT and it is constructed correctly with Network Diagram, calculation and with very detailed explanation (*how it is estimated and calculated*) on most of the components in the Network Diagram. The cost is estimated via COCOMO correctly.	Critical path is analysed using PERT and it is constructed partially correct with Network Diagram, calculation and limited explanation on how each component in the Network Diagram was estimated and calculated. The cost is estimated via COCOMO correctly.	Critical path is analysed using PERT and it is constructed partially correct with Network Diagram, calculation and limited explanation on how each component in the Network Diagram was estimated and calculated. The cost is estimated via COCOMO but have errors.	Critical path is analysed using PERT, but it is not constructed, or there is no critical path/PERT and their analysis. There is no or limited explanation on the components in the Network Diagram. The cost is estimated via COCOMO but have errors or there is no costing involved.

Category & Expectation	Outstanding (85 - 100%)	Excellent (70 - 84%)	Competent (60 - 69%)	Satisfactory (50 - 59%)	Unsatisfactory (0 – 49%)
Code snippet (20 marks) & Expect to create a self-explanatory and easy to read code snippets.	≥ 85% of the (*Java*) code snippets are self-explanatory, easy to read, cover all the core functions, and do not have any code issues. All relevant code (*including all the test cases*) is committed by the members working on their assigned user stories in GitHub. The submitted Eclipse compressed project must be able to be uncompressed and imported into Eclipse on the Percy Gee Lab Computer without any error. There is a very detailed README file on how to uncompress, load from the computer in Percy Gee laboratory, compile and run all the code and test cases. There is also a working public GitHub link to the project.	≥ 70% of the (*Java*) code snippets are self-explanatory, easy to read and cover all the core functions, and do not have any code issues. All relevant code (*including all the test cases*) is committed by the members working on their assigned user stories in GitHub. The submitted compressed Eclipse project must be able to be uncompressed and imported into Eclipse on the Percy Gee Lab Computer without any error. There is a detailed README file on how to uncompress, load from the computer in Percy Gee laboratory, compile and run all the code and test cases. There is also a working public GitHub link to the project.	≥ 60% of the (*Java*) code snippets are self-explanatory, easy to read and cover all the core functions, and do not have any code issues. All relevant code (*including all the test cases*) is committed by the members working on their assigned user stories in GitHub. The submitted compressed Eclipse project must be able to be uncompressed and imported into Eclipse on the Percy Gee Lab Computer without any error. There is a somewhat detailed README file on how to uncompress, load from the computer in Percy Gee laboratory, compile and run all the code and test cases. There is also a working public GitHub link to the project.	≥ 50% of the (*Java*) code snippets are self-explanatory, easy to read and cover all the core functions, and do not have any code issues. Some (*incomplete*) relevant code (*including all the test cases)* is committed by the members working on their assigned user stories in GitHub. The submitted compressed Eclipse project must be able to be uncompressed and imported into Eclipse on the Percy Gee Lab Computer without any error. There is a less detailed README file on how to uncompress, load from the computer in Percy Gee laboratory, compile and run all the code and test cases. There is also a working public GitHub link to the project.	< 50% of the (*Java*) code snippets are self-explanatory, easy to read and cover all the core functions, and do not have any code issues. Limited relevant code (*including all the test cases*) is committed by the members working on their assigned user stories in GitHub. The submitted compressed Eclipse project could not be uncompressed and imported into Eclipse on the Percy Gee Lab Computer without any error. There is no detailed README file on how to uncompress, load from the computer in Percy Gee laboratory, compile and run all the code and test cases. There is no working public GitHub link to the project.

Category & Expectation	Outstanding (85 - 100%)	Excellent (70 - 84%)	Competent (60 - 69%)	Satisfactory (50 - 59%)	Unsatisfactory (0 – 49%)
Black-Box and White-Box Testing and Coverage (20 marks) & Expect to analyse the requirements and create test cases to evaluate the system.	≥ 85% of the test cases passed. ≥ 85% of the test cases are documented properly indicating what test techniques are used, what testing tools are used and the expected and actual test results. The average test coverage for all the black-box and white-box testing coverage is ≥ 85%. The black-box test cases must be in a package "*test.blackbox*" with clear comments in all the black-box test cases code explaining which black-box testing technique is used for which black-box test cases. The black-box test cases must be able to be executed in Eclipse's Junit showing the test coverage on the Percy Gee Lab Computer without any error. The white-box test cases must be in a package	≥ 70% of the test cases passed. ≥ 70% of the test cases are documented properly indicating what test techniques are used, what testing tools are used and the expected and actual test results. The average test coverage for all the black-box and white-box testing coverage is ≥ 70%. The black-box test cases must be in a package "*test.blackbox*" with clear comments in all the black-box test cases code explaining which black-box testing technique is used for which black-box test cases. The black-box test cases must be able to be executed in Eclipse's Junit showing the test coverage on the Percy Gee Lab Computer without any error.	≥ 60% of the test cases passed. ≥ 60% of the test cases are documented properly indicating what test techniques are used, what testing tools are used and the expected and actual test results. The average test coverage for all the black-box and white-box testing coverage is ≥ 60%. The black-box test cases must be in a package "*test.blackbox*" with clear comments in all the black-box test cases code explaining which black-box testing technique is used for which black-box test cases. The black-box test cases must be able to be executed in Eclipse's Junit showing the test coverage on the Percy Gee Lab Computer without any error.	≥ 50% of the test cases passed. ≥ 50% of the test cases are documented properly indicating what test techniques are used, what testing tools are used and the expected and actual test results. The average test coverage for all the black-box and white-box testing coverage is ≥ 50%. The black-box test cases must be in a package "*test.blackbox*" with clear comments in all the black-box test cases code explaining which black-box testing technique is used for which black-box test cases. The black-box test cases must be able to be executed in Eclipse's Junit showing the test coverage on the Percy Gee Lab Computer without any error.	< 50% of the test cases passed. < 50% of the test cases are documented properly indicating what test techniques are used, what testing tools are used and the expected and actual test results. The average test coverage for all the black-box and white-box testing coverage is < 50%. The black-box test cases are not in a package "*test.blackbox*" and there are no clear comments in all the black-box test cases code explaining which black-box testing technique is used for which black-box test cases. The black-box test cases could not be executed in Eclipse's Junit showing the test coverage on the Percy Gee Lab Computer.

Category & Expectation	Outstanding (85 - 100%)	Excellent (70 - 84%)	Competent (60 - 69%)	Satisfactory (50 - 59%)	Unsatisfactory (0 – 49%)
	"*test.whitebox*" with clear comments in the white-box test cases code explaining which white-box testing technique is used for which white-box test cases. The white-box test cases must be able to be executed in Eclipse's Junit showing the test coverage on the Percy Gee Lab Computer without any error.	The white-box test cases must be in a package "*test.whitebox*" with clear comments in the white-box test cases code explaining which white-box testing technique is used for which white-box test cases. The white-box test cases must be able to be executed in Eclipse's Junit showing the test coverage on the Percy Gee Lab Computer without any error.	The white-box test cases must be in a package "*test.whitebox*" with clear comments in the white-box test cases code explaining which white-box testing technique is used for which white-box test cases. The white-box test cases must be able to be executed in Eclipse's Junit showing the test coverage on the Percy Gee Lab Computer without any error.	The white-box test cases must be in a package "*test.whitebox*" with clear comments in the white-box test cases code explaining which white-box testing technique is used for which white-box test cases. The white-box test cases must be able to be executed in Eclipse's Junit showing the test coverage on the Percy Gee Lab Computer without any error.	The white-box test cases must be in a package "*test.whitebox*" with clear comments in the white-box test cases code explaining which white-box testing technique is used for which white-box test cases. The white-box test cases could not be executed in Eclipse's Junit showing the test coverage on the Percy Gee Lab Computer.
Research (10 marks) & Expect to critically evaluate Software Quality related research papers.	The research report presents a very clear description of the underlying mechanism, imitations and improvements. The group understands the paper very well.	The research report presents a good description of the underlying mechanism, and limitations and improvements.	The research report presents a satisfactory description of the underlying mechanism, and limitations and improvements. However, some of the parts are a bit unclear or does not make sense.	The research report presents with less clear description of the underlying mechanism, limitations and improvements. However, the report shows some understanding of the paper by the group.	There is limited description of the underlying mechanism of the chosen research paper, limitations and improvements.
Presentation (20 marks) & Expect to clearly summarize and present the features of the system.	≥ 85% of the presentation is very organized and is very easy to follow. Transitions between group members were well planned and executed cleanly.	≥ 70% of the presentation is very organized and is very easy to follow. Transitions might have been slightly discontinuous but did not take away greatly	≥ 60% of the presentation is very organized and is very easy to follow. Transitions might have been slightly discontinuous but did not take away greatly	≥ 50% of the presentation is very organized and is very easy to follow. Transitions between members are jumpy or awkward.	< 50% of the presentation is very organized and is very easy to follow. Transitions between members are jumpy or awkward.

Category & Expectation	Outstanding (85 - 100%)	Excellent (70 - 84%)	Competent (60 - 69%)	Satisfactory (50 - 59%)	Unsatisfactory (0 – 49%)
	Visual aids are used effectively throughout the presentation. Group members use visual aids as a supplement, not as a crutch.	from the overall presentation. Visual aids are somewhat effective but were not used consistently throughout the presentation.	from the overall presentation. Visual aids are somewhat effective but were not used consistently throughout the presentation.	Visual aids are somewhat effective but were not used consistently throughout the presentation.	Visual aids used do not support the verbal presentation.

5. Marking and Feedback

- TAs and the module conveners will work together to mark all submissions.
- TA will help in distributing feedback.
- A paragraph of "Overall comment" will be given at the end of the feedback.

6. Cover Sheet of Academic Integrity

Module Code: _____

Assignment: _____

I understand that this is a piece of coursework. I confirm that I handed in a signed Declaration of Academic Honesty Form (available at https://campus.cs.le.ac.uk/ForStudents/plagiarism/) and that I am fully aware of the statements contained therein. I understand if the submission will be checked by Turnitin Software. Any submission with similarity scores greater than approximately 20% may be further assessed individually and may be reported to Plagiarism Office. You should aim to keep the similarity score to less than 10%.

	Member 1	Member 2	Member 3	Member 4	Member 5
Last, First Name: (in CAPITALS)					
Student ID: (e.g., ab123)					
Date:					
Signature:					

7. Resources

- Agile Scrum development: https://www.atlassian.com/agile/scrum.
- Development tools:
  - GitHub: https://github.com.
  - Trello: https://trello.com/en.
  - Eclipse: https://www.eclipse.org.
- Video compress tool: https://www.veed.io/tools/video-compressor.
- Report templates:
  - IEEE Xplore: https://ieeexplore-ieee-org.ezproxy3.lib.le.ac.uk/Xplore/home.jsp
  - Science Direct: https://www.sciencedirect.com/
  - Scopus: https://www-scopus-com.ezproxy3.lib.le.ac.uk/search/form.uri?display=basic#basic
- Textbooks (*non-exhaustive*)
  - Software quality assurance consistency in the face of complexity and change, by Neil Walkinshaw
  - Software Testing and Analysis - Process, Principles, and Techniques, by Mauro Pezze and Michal Young
- Research Papers (*non-exhaustive*)
  - EVOSUITE at the SBST 2021 Tool Competition, by S. Vogl, S. Schweikl, G. Fraser, A. Arcuri, J. Campos, and A. Panichella
  - A Tutorial on Using and Extending the EvoSuite Search-Based Test Generator, by G. Fraser
  - Java Enterprise Edition Support in Search-Based JUnit Test Generation, by A. Arcuri and G. Fraser
  - Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins, by A. Arcuri, J. Campos, and G. Fraser
  - Search-based Data-flow Test Generation, by M. Vivanti, A. Mis, A. Gorla, and G. Fraser
  - Mutation-driven Generation of Unit Tests and Oracles, by G. Fraser and A. Zeller

The A-Z of databases and electronic resources can be found at the URL of https://le.ac.uk/library/search-collections/databases-az

8. CO7095 Project List

1) Multiple choice question and answer desktop application.
2) A Credit scoring tool.
3) A Financial investment assistant.
4) A Job Recruiter Assistant Chatbot.
5) A Crowdsourcing Tool for Route Optimisation.
6) A Timetable Management Tool.
7) A film recommendation system.
8) A mobile app for campus navigation.
9) A platform for evaluating schools.
10) Football league scheduling.
11) Flight Journeys System.
12) Job Finder Web or Mobile App.
13) Mobile Client for Asset Management System.
14) Multidisciplinary Project Management System.
15) News monitoring and analytics platform.
16) Smart Shopping Trolley.
17) Software Support for Car Sharing.
18) Stock Market Trading System.
19) Takeaway Menu System.
20) Unit Job Scheduling.
21) Virtual Chat Assistant.
22) A tool to manage pension benefits.
23) Medical Screening Assistant.
24) Placements Management Web App.

25) Social Network Data Capture and Analysis Tool.

26) A tool for tracking placement applications, approvals, assessments, and visits.

27) An application for managing chronic patients' health.

28) Ticket booking system for mobile devices.

29) A scientific conference organisation assistant.

30) A tool for managing software projects.

31) Software Bug Tracking and Reporting Tool.

32) Examination paper management system.

33) Clinical research management tool.

34) Software Code Review Tool.

35) Ubiquitous Mobile Music Player Application.

36) Text/Code Editor/Compiler.

37) An Automated Collect and Analysis Tool for Real Estate.

38) Stock Control System.

39) Teach-SPL: A teaching tool for Software Product Lines.

40) A judge system for programming competitions.

9. Automatic Code Generation Research Papers

1) MPCoder: Multi-user Personalized Code Generator with Explicit and Implicit Style Representation Learning, ACL 2024
2) StepCoder: Improving Code Generation with Reinforcement Learning from Compiler Feedback, ACL 2024
3) IRCoder: Intermediate Representations Make Language Models Robust Multilingual Code Generators, ACL 2024
4) Who Wrote this Code? Watermarking for Code Generation, ACL 2024
5) MapCoder: Multi-Agent Code Generation for Competitive Problem Solving, ACL 2024
6) ArchCode: Incorporating Software Requirements in Code Generation with Large Language Models, ACL 2024
7) CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges, ACL 2024
8) Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models, ACL 2024
9) Iterative Refinement of Project-Level Code Context for Precise Code Generation with Compiler Feedback, ACL Findings 2024
10) DevEval: A Manually-Annotated Code Generation Benchmark Aligned with Real-World Code Repositories, ACL Findings 2024
11) Functional Overlap Reranking for Neural Code Generation, ACL Findings 2024
12) Comments as Natural Logic Pivots: Improve Code Generation via Comment Perspective, ACL Findings 2024
13) OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement, ACL Findings 2024
14) Evaluating In-Context Learning of Libraries for Code Generation Arkil, NAACL 2024
15) Exploring Language Model's Code Generation Ability with Auxiliary Functions, NAACL 2024
16) Text-to-Code Generation with Modality-relative Pre-training, EACL 2024
17) L2MAC: Large Language Model Automatic Computer for Extensive Code Generation, ICLR 2024
18) Is Self-Repair a Silver Bullet for Code Generation?, ICLR 2024

19) CodeChain: Towards Modular Code Generation Through Chain of Self-revisions with Representative Sub-modules, ICLR 2024
20) CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models, ICSE 2024
21) Evaluating Large Language Models in Class-Level Code Generation, ICSE 2024
22) Beyond Code Generation: An Observational Study of ChatGPT Usage in Software Engineering Practice, FSE 2024
23) ClarifyGPT: A Framework for Enhancing LLM-based Code Generation via Requirements Clarification, FSE 2024
24) DeciX: Explain Deep Learning Based Code Generation Applications, FSE 2024
25) Do Large Language Models Pay Similar Attention Like Human Programmers When Generating Code?, FSE 2024
26) PPM: Automated Generation of Diverse Programming Problems for Benchmarking Code Generation Models, FSE 2024
27) Python Code Generation by Asking Clarification Questions, ACL 2023
28) CodeIE: Large Code Generation Models are Better Few-Shot Information Extractors, ACL 2023
29) ReCode: Robustness Evaluation of Code Generation Models, ACL 2023
30) Code4Struct: Code Generation for Few-Shot Event Structure Prediction, ACL 2023
31) Natural Language to Code Generation in Interactive Data Science Notebooks, ACL 2023
32) Self-Edit: Fault-Aware Code Editor for Code Generation, ACL 2023
33) Modular Visual Question Answering via Code Generation, ACL 2023
34) Aligning Offline Metrics and Human Judgments of Value for Code Generation Models, ACL Findings 2023
35) A Simple, Yet Effective Approach to Finding Biases in Code Generation, ACL Findings 2023
36) Personalized Distillation: Empowering Open-Sourced LLMs with Adaptive Learning for Code Generation, EMNLP 2023
37) Symbolic Planning and Code Generation for Grounded Dialogue, EMNLP 2023
38) CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code, EMNLP 2023
39) API-Assisted Code Generation for Question Answering on Varied Table Structures, EMNLP 2023

40) On Sample-Efficient Code Generation, EMNLP 2023
41) Execution-Based Evaluation for Open-Domain Code Generation, EMNLP 2023
42) Syntax-Aware Retrieval Augmented Code Generation, EMNLP 2023
43) Symbolic Planning and Code Generation for Grounded Dialogue, EMNLP 2023
44) CodeGen4Libs: A Two-Stage Approach for Library-Oriented Code Generation, ASE 2023
45) From Misuse to Mastery: Enhancing Code Generation with Knowledge-Driven AI Chaining, ASE 2023
46) On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot, ICSE 2023
47) SkCoder: A Sketch-based Approach for Automatic Code Generation, ICSE 2023
48) Towards Greener Yet Powerful Code Generation via Quantization: An Empirical Study, FSE 2023
49) Coder Reviewer Reranking for Code Generation, ICML 2023
50) DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation, ICML 2023
51) Outline, Then Details: Syntactically Guided Coarse-To-Fine Code Generation, ICML 2023
52) LEVER: Learning to Verify Language-to-Code Generation with Execution, ICML 2023
53) Uncovering and Quantifying Social Biases in Code Generation, NeurIPS 2023
54) Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation, NeurIPS 2023