# Yet another talk on generative AI

*Experiences of a non-expert*

Luigi Lavazza
Università degli Studi dell'Insubria, Varese, Italy

# Luigi Lavazza

**Professional experience**

- Professor of Computer Science at the University of Insubria at Varese, Italy.
- Scientific consultant in digital innovation projects at CEFRIEL – Politecnico di Milano.

**Scientific Activity**

- Research: Empirical software engineering, software metrics and software quality evaluation; project management and effort estimation; Software process modeling, measurement and improvement; Open Source Software.
- Several international research projects
- Reviewer of EU funded projects.
- Co-author of over 180 scientific articles.
- PC member of several international Software Engineering conferences
- Editor in chief of the IARIA International Journal On Advances in Software (2013-2018).
- IARIA fellow since 2011

# Disclaimer

- I am not an AI expert

- Having heard a lot about Generative Artificial Intelligence (GAI), and having heard a lot of contradictory statements, I decided to try it
- Here I report my experience with GAI
- In a domain (scientific publishing) in which I am an expert

# Objective (1)

- Being a computer scientist (a software engineer) I decided to test GAI as an instrument for writing a scientific paper.
- The idea was to let GAI do (almost) all the work.
  - With manual integration and adjustments, where necessary
- Note: the goal was NOT to write a completely fake paper.
- The goal (at least initially) was to produce a reasonable paper.
  - Maybe a paper that just presents differently already known facts.

# Objective (2)

- What kind of paper?
- In my research area (empirical software engineering) Systematic Literature Reviews (SLR) are quite popular.
- Characteristics: no new contents, just an overview of what is available, what techniques are most used, what datasets are employed, what statistical or ML methods are used, etc., what are the merits and achievements of the published papers, etc.

- Going for a SLR seemed a good idea, since GAI did not need to produce anything really new from a scientific point of view. In other words, I chose *a relatively easy task*.

# Target level (where to submit)

- A conference, because I needed a response in predictable and short time.
- A mid-quality conference
  - A high-level one, with very low acceptance rate would have been a too high target: a rejection would prove nothing
  - Not a low-level one, i.e., one of those conference that accept practically all submitted papers, because acceptance there would not prove anything.
- I will not disclose the identity of the conference, however I can tell that according to the Italian academic ranking system it is at the same level as ICSEA.

# The target conference and AI

- An interesting feature of the target conference is that they have clear politics concerning the usage of GAI:

## On use of AI (Artificial Intelligence) or AI-assisted technologies in research papers

Inspired by Elsevier's and ICML (International Conference on Machine Learning) policies regarding the use of AI or AI-assisted technologies, such as Large Language Models, the [redacted] 2023 - Research Track 2023 tentatively adopted a policy to be strictly followed by the submitted research papers. For the next editions of [redacted] Research Track, the upcoming [redacted] policy will take place.

What is forbidden:

- Text or image produced/generated entirely by AI or AI-assisted technologies; and
- Any AI or AI-assisted technology as an author.

Using GAI as I did was not allowed!

What is allowed:

- Using AI or AI-assisted technologies for editing or polishing author-written text;
- Using AI or AI-assisted technologies for improving the quality of images regarding contrast and clarity; and
- Investigating the use of AI or AI-assisted technologies to support Software Engineering activities.

# Topic of the SLR

- I needed a topic that
  - I know sufficiently well, to understand if the generated text is OK or not.
  - There are enough publications to support a SLR
  - There are not too many publications
    - otherwise the selection of relevant paper could be too difficult
  - There are no SLR already available.
    - To avoid comparisons with existing SLRs

# Topic of SLR

- I chose **Self-Admitted Code Smells (SACS)**

- "Self-Admitted Code Smells" on google: 0 papers [May 6th, 2023]
  - In fact, most researchers wrote about self-admitted technical debt
  - But there is a strict connection between technical debt and code smell

- By the way, a code smell is a symptom that code probably has problems (likely faultiness, difficult to maintain, etc.)

# Which GAI tool?

- I used chat GPT
  - just because it is the most famous GAI available.
- I used the free version.

# How to approach the task

- A naïve approach: let’s ask GAI to do everything

- But not so naïve:
  - let’s give GAI a thorough explanation of what is desired.

# First step: the request

- I issued a quite detailed and precise request, specifying
  - The type of paper needed (SLR)
  - The topic
  - The guidelines that are typically used by humans for carrying out a SLR
  - The sources to be used
  - What should be contained in the SLR
  - The desired output format (LaTeX)

# First request (1)

Please, I need a systematic literature review in the form of a scientific paper.

The review has to concern self-admitted code smells in software code.

The systematic literature review must follow the directives provided by Kitchenham and Charters in the paper "Guidelines for performing systematic literature reviews in software engineering version 2.3".

Relevant articles must be selected from IEEExplore, scholar google and the ACM repositories.

# First request (2)

The review should illustrate the number of relevant publications per year and the publication venues.

The review should focus on: the definition of self-admitted code smell, the techniques that have been reported in the literature for the detection of self-admitted code smells, the effectiveness of such techniques, the practical consequences on software development.

The review should contain the list of relevant primary studies.

If possible, the review should be delivered as a Latex file.

# Note

- The allowed answer has a maximum length.
- Thus, it was necessary to prompt the tool for more.
- The tool obeyed.
- This was just annoying: an answer could end in the middle of a word and the rest of that word was provided after prompting for more. Example

comments and use them to identify SACS. Textual analysis techniques use natural language processing (NLP) to identify SACS in comments or commit messages. Machine learning techniques use supervised or unsup
User
ok, please go on.
ChatGPT

3.3 SACS Detection Techniques (continued)
ervised learning algorithms to train a model to identify SACS. Pattern recognition techniques use predefined patterns or rules to identify SACS in code or comments. The review found that the most commonly used technique for SACS

- In what follows I present the complete outcome, without interruptions.

# What was produced: Abstract

Self-admitted code smells are coding structures that developers include in the code intentionally to communicate their awareness of a potential problem, but they fail to address the issue due to time constraints, lack of knowledge or priority. Such code smells are hard to detect using automatic tools, and their presence can negatively impact the software quality and maintainability. In this paper, we perform a systematic literature review to identify and analyze the current state-of-the-art techniques for detecting self-admitted code smells. We searched three leading digital libraries, IEEE Explore, Scholar Google, and the ACM repository, and identified a total of 45 relevant studies published between 2010 and 2022. The review identified several techniques that have been proposed for the detection of self-admitted code smells, such as textual analysis, machine learning, and pattern recognition. We analyzed the effectiveness of these techniques, their practical implications on software development, and the research gaps. Our findings indicate that existing detection techniques have high accuracy rates but are limited by their inability to detect context-specific code smells. Moreover, the results show that self-admitted code smells have practical implications on software development, including code review, refactoring, and testing. Our review provides a comprehensive understanding of the current state-of-the-art techniques for detecting self-admitted code smells and their practical implications on software development.

# Comment on the abstract

- Very good.
- Well structured:
  - Background
  - Aim
  - Method
    - It states that 45 relevant studies published between 2010 and 2022 were retrieved and surveyed
  - Results

# What was produced: Keywords

- Self-Admitted Code Smells, Systematic Literature Review, Code Smell Detection, Software Quality, Maintainability

# Comment on the keywords

- Good
  - Most words were in the request.
  - *Software Quality* and *Maintainability* were introduced by GAI

# What was produced: Introduction

Self-admitted code smells (SACS) are coding structures that developers include in the code to indicate their awareness of potential issues, but they fail to address the problem due to time constraints, lack of knowledge or priority. SACS can negatively impact software quality and maintainability, and they are challenging to detect using automated tools since they are intentional. Therefore, manual detection of SACS is necessary to improve software quality and maintainability. In this paper, we perform a systematic literature review to identify and analyze the current state-of-the-art techniques for detecting SACS.

# Comment on the introduction

- Poor
  - Definitely too short (shorter than the abstract!)
  - A very brief description of the topic (SACS) and the associated danger
  - Focus on the detection of SACS

# What was produced: Methodology

We followed the guidelines provided by Kitchenham and Charters [1] for performing systematic literature reviews in software engineering. We searched three leading digital libraries, IEEE Explore, Scholar Google, and the ACM repository, for publications relevant to SACS. We included publications written in English and published between 2010 and 2022. The search keywords included "self-admitted code smells," "code smell detection," "software quality," and "maintainability." The search yielded a total of 86 publications. After applying the inclusion and exclusion criteria, we identified 45 relevant publications.

# Comment on the methodology section

- Poor
  - Definitely too short
  - It does not add any valuable information with respect to what was said in the abstract
    - The only new piece of info is that 86 potentially relevant papers were initially retrieved.

# What was produced: Results

## 3.1 Yearly Distribution of Relevant Publications

Figure 1 shows the yearly distribution of relevant publications. The results indicate that the research interest in SACS detection has been growing steadily since 2010, with a peak in 2019 and a decline in 2020 and 2021.

Insert Figure 1 here

# What was produced: Results

## 3.2 Publication Venues

Table 1 shows the publication venues of the identified publications. The results indicate that most of the publications are published in software engineering conferences and journals, such as the IEEE Transactions on Software Engineering and the International Conference on Software Engineering.

Insert Table 1 here

# What was produced: Results

## 3.3 SACS Detection Techniques

The literature review identified several techniques that have been proposed for the detection of SACS, such as textual analysis, machine learning, and pattern recognition. These techniques aim to extract features from the code or comments and use them to identify SACS. Textual analysis techniques use natural language processing (NLP) to identify SACS in comments or commit messages. Machine learning techniques use supervised or unsupervised learning algorithms to train a model to identify SACS. Pattern recognition techniques use predefined patterns or rules to identify SACS in code or comments. The review found that the most commonly used technique for SACS detection is textual analysis, followed by machine learning and pattern recognition.

# What was produced: Results

## 3.4 Effectiveness of SACS Detection Techniques

The literature review analyzed the effectiveness of SACS detection techniques. The results indicate that the existing techniques have high accuracy rates in detecting SACS. For instance, some studies reported accuracy rates of up to 95%. However, the studies also found that the techniques are limited by their inability to detect context-specific SACS. Moreover, some studies reported that the techniques may generate false positives or false negatives, which can lead to incorrect conclusions.

# What was produced: Results

## 3.5 Practical Implications on Software Development

The literature review analyzed the practical implications of SACS on software development. The results indicate that SACS have practical implications on software development, including code review, refactoring, and testing. SACS can help developers identify potential issues in the code and prioritize the necessary actions to address the issues. Moreover, the review found that SACS can increase the efficiency of code review by highlighting the problematic areas in the code. Additionally, SACS can guide the refactoring process by indicating the areas that require attention.

# Comments on the Results section

- Poor

- Too short
- Shallow
- No reference to the primary papers, i.e., the sources
    - this is not acceptable for a SLR!
- Tables and figures were not provided.

# What was produced: Discussion

The literature review provides a comprehensive understanding of the current state-of-the-art techniques for detecting SACS and their practical implications on software development. The review identified several techniques that have been proposed for SACS detection, such as textual analysis, machine learning, and pattern recognition. The results indicate that the existing techniques have high accuracy rates in detecting SACS but are limited by their inability to detect context-specific SACS. Moreover, SACS have practical implications on software development, including code review, refactoring, and testing.

# Comments on the Results section

- Poor

- Not a real discussion.
- A sort of very high-level abstract, more suitable for a conclusions section than for a discussion.
- As usual, too short (one paragraph).

# What was produced: Conclusion

In conclusion, the literature review provides a comprehensive understanding of the current state-of-the-art techniques for detecting SACS and their practical implications on software development. The review identified several techniques that have been proposed for SACS detection, analyzed their effectiveness, and discussed their practical implications on software development. The results indicate that SACS detection is an important area of research that can help improve software quality and maintainability. However, the research gaps and limitations of the existing techniques suggest the need for further research in this area.

# Comments on the Conclusion section

- Acceptable

- A brief summary of results.
- At the correct level of abstraction.
- As usual, too short, even for a conclusive section.

# What was produced: References

[1] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering version 2.3," Keele University, UK, 2007.

# Comments on the References

- Unacceptable

- Just one reference
- No reference at all concerning primary papers

# Evaluation of the first output from GAI

- Too short
- Makes reference to a table and a figure not given
- Contents too shallow
- No references, except the one mentioned in the request
- No related work section
- Not in LaTex

- In practice, what GAI produced was **VERY FAR** from credible scientific paper.

# Step 2

- To improve the text produced by GAI, I issued some specific requests

- I asked for references.
- GAI complied, providing 10 references at a time.

# Please give me the references


Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., & Zhou, Y. (2010). Evaluating the effectiveness of SACA-based program analysis tools. In Proceedings of the 7th int. conf. on software engineering and formal methods (pp. 15-24). Springer.

Bavota, G., Oliveto, R., Di Penta, M., & De Lucia, A. (2013). An empirical study on self-reported technical debt. In Proceedings of the 2013 international conference on software engineering (pp. 715-724). IEEE Press.

Fontana, F. A., & Braione, P. (2013). An experimental comparison of machine learning techniques for SACS detection. Journal of Systems and Software, 86(5), 1256-1266.

Fontana, F. A., & Zanoni, M. (2012). Toward a general definition of self-admitted technical debt: Preliminary results. In Proceedings of the 2012 international workshop on managing technical debt (pp. 23-30). IEEE Press.

Garcia, A., & Zaidman, A. (2013). Using latent dirichlet allocation for SACS detection. In Proceedings of the 2013 international conference on software engineering (pp. 732-741). IEEE Press.

Kamei, Y., Shihab, E., & Adams, B. (2013). Studying the impact of SACS on software quality. In Proceedings of the 2013 international conference on software engineering (pp. 592-601). IEEE Press.

Moha, N., Guéhéneuc, Y. G., Duchien, L., & Le Meur, A. F. (2010). DECOR: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering, 36(1), 20-36.

Palomba, F., Panichella, A., & Zaidman, A. (2017). A text mining approach for the detection of self-admitted technical debt. Empirical Software Engineering, 22(6), 3167-3212.

Shihab, E., & Hassan, A. E. (2013). What do we know about SACS? An exploratory study. In Proceedings of the 2013 international conference on software maintenance (pp. 480-483). IEEE Press.

Zhou, Y., & Leung, H. (2014). Toward a definition of code smells for JavaScript. Journal of Systems and Software, 94, 56-69.

# More references, please


Azadi, S., Mirshokraie, S. A., & Monden, A. (2015). Identifying self-admitted technical debt in open source projects using text mining. Journal of Systems and Software, 101, 101-116.

Bavota, G., & Russo, B. (2014). Investigating the role of SACS through the history of two software projects. Journal of Software: Evolution and Process, 26(6), 556-577.

Chatzigeorgiou, A., Manakos, A., & Stephanides, G. (2018). Investigating the distribution of SACS in large-scale software projects. Journal of Systems and Software, 140, 126-141.

Idri, A., & Abran, A. (2014). Towards a better understanding of technical debt in agile software development. Journal of Software: Evolution and Process, 26(7), 676-693.

Jiang, B., Adams, B., & Hassan, A. E. (2013). Detecting and characterizing SACS in software systems. In Proceedings of the 10th working conference on mining software repositories (pp. 41-50). IEEE Press.

Khomh, F., & Vaucher, S. (2012). Gaining insight into SACS using association rule mining. In Proceedings of the 2012 international workshop on principles of software evolution (pp. 61-70). ACM.

Li, Z., & Shatnawi, R. (2014). An exploratory study of technical debt in mobile apps. Journal of Systems and Software, 93, 121-130.

Palomba, F., Panichella, A., Proksch, S., & Zaidman, A. (2019). A machine learning approach for SACS detection. IEEE Transactions on Software Engineering, 45(3), 254-280.

Potdar, A., & Shihab, E. (2014). An exploratory study on the evolution of SACS. In Proceedings of the 2014 international conference on software maintenance and evolution (pp. 91-100). IEEE Press.

Tsantalis, N., & Chatzigeorgiou, A. (2011). Identification of SACS using gene expression programming. In Proceedings of the 2011 international symposium on empirical software engineering and measurement (pp. 19-28). IEEE Press.

# GAI explains

- User:

  in the abstract, you wrote that 45 relevant studies were identified. Please, list them.

- ChatGPT:

  I apologize for the confusion, but **I did not actually identify 45 relevant studies in the abstract**. I used "45" as an example number to illustrate how many relevant studies could be identified in a systematic literature review.

  In reality, the number of relevant studies identified in the review will depend on several factors such as the search terms used, the inclusion and exclusion criteria, and the quality of the studies themselves.

  For this reason, it would not be appropriate or feasible to provide a list of 45 relevant studies without actually conducting the review. The list of relevant studies will be presented in the "Results" section of the review after the selection and screening process has been completed.

# Creation of missing figure

- User:

Please, considering the 20 references you gave me previously, provide a figure to be used as figure 1, reporting the frequency of publications per year.

- ChatGPT

I apologize for any confusion. As an AI text-based model, I'm unable to generate visual figures directly. However, I can help you with the data needed to create the figure you mentioned. Here is the frequency of publications per year based on the 20 references provided:

2010: 1 publication
2011: 1 publication
2012: 2 publications
2013: 6 publications
[...]

Please note that these numbers are based on the references provided earlier, and there may be additional relevant publications that were not included in the list. You can use this data to create a bar chart or any other visualization representing the frequency of publications per year.

# Comments on the references

- **Absolute disaster**

- The provided references are not real: GAI <u>invented</u> them

This is way too much *<u>generative</u>*!

- But the data to create the figure were consistent with the made-up references!

# Step 2 - Related work

- I asked for related work

- User
  - Please write a related work section, including also references to reviews dealing with code smell in general, not just self-admitted ones.
- ChatGPT
  - Sure, here's a sample related work section that covers reviews of code smells in general, as well as those specifically related to self-admitted code smells:

# Related Work

Code smells have been widely studied in the software engineering literature, and several reviews have been conducted to summarize the state of the art in this area. For instance, Marinescu [1] provides a comprehensive overview of code smells and their impact on software quality, while Moha et al. [2] present a survey of detection and refactoring techniques for code smells.

More recently, researchers have focused on a specific type of code smell known as self-admitted code smells (SACS), which are code smells that developers explicitly acknowledge in the code comments or documentation [3]. Several studies have investigated SACS in the context of software maintenance and evolution [4, 5], and proposed various techniques for their detection and refactoring [6, 7].

To the best of our knowledge, no systematic literature review has been conducted to synthesize the existing research on SACS detection and its practical implications on software development. Therefore, this paper aims to fill this gap by following the guidelines proposed by Kitchenham and Charters [8] for conducting a systematic literature review in software engineering.

# Related work references


[1] Marinescu, R. (2016). Detection strategies: Metrics-based rules for detecting design flaws in OO software. CRC Press.

[2] Moha, N., Nayrolles, M., Palma, F., & Kessentini, M. (2017). A survey on code smells detection techniques. Journal of Systems and Software, 123, 124-151.

[3] Yamashita, A., Moonen, L., & Kamei, Y. (2013). Are all code smells harmful? A study of God classes and Brain classes in the evolution of three open source systems. In Proceedings of the 2013 international conference on software engineering (pp. 762-771). IEEE Press.

[4] Azadi, S., Mirshokraie, S. A., & Monden, A. (2015). Identifying self-admitted technical debt in open source projects using text mining. Journal of Systems and Software, 101, 101-116.

[5] Potdar, A., & Shihab, E. (2014). An exploratory study on the evolution of self-admitted technical debt. In Proceedings of the 2014 international conference on software maintenance and evolution (pp. 91-100). IEEE Press.

[6] Palomba, F., Panichella, A., Proksch, S., & Zaidman, A. (2019). A machine learning approach for the detection of self-admitted technical debt. IEEE Transactions on Software Engineering, 45(3), 254-280.

[7] Khomh, F., & Vaucher, S. (2012). Gaining insight into self-admitted technical debt using association rule mining. In Proceedings of the 2012 international workshop on principles of software evolution (pp. 61-70). ACM.

[8] Kitchenham, B., & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering version 2.3. Journal of Software Engineering Research and Development, 1(1), 1-26.

# Comments on the Related work section

- Another disaster

- The text is too short
  - It does not cover most work done in code smell research
- The provided references are not real: GAI invented them as well

# Asking GAI about invented references

- Among the references you provided for the related work section, there are many that do not correspond to any real paper. Fort instance, Moha, Nayrolles, Palma and Kessentini never wrote a paper titled "A survey on code smells detection techniques." Can you please provide a reference section that makes reference to real papers? You could use scholar.google.com to retrieve references to real papers on code smells.
- ChatGPT
  - Apologies for the confusion caused by the incorrect references. I appreciate your understanding. I will now provide a revised reference section for the related work, based on real papers on code smells retrieved from Google Scholar:
    - Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley.
    - …
  - Please note that this is not an exhaustive list, but it includes influential papers that cover various aspects of code smells. You can retrieve the complete citations for these papers from Google Scholar or the respective publication venues for further reference.

# Report about real related work

  - Good. Please, provide a few lines summarizing the contributions of these papers. You may look at the abstracts of these papers.
- ChatGPT
  - Certainly! Here are brief summaries of the contributions of the selected papers:
    - Fowler (1999): "Refactoring: Improving the Design of Existing Code" introduces the concept of refactoring as a technique to improve code quality, maintainability, and design. It presents various refactoring patterns and guidelines for identifying and addressing code smells.
    - …
  - Please note that these are only brief summaries, and each paper contributes much more in-depth insights and findings within their respective domains. For a comprehensive understanding, I recommend referring to the full papers and their abstracts.

# Ad hoc references

- I wanted to report in the related work section any paper written by members of the Program Committee
  - Because of course members of the PC would recognize their papers and notice if there is something wrong in the way they are mentioned
- Hence, I asked GAI to provide such references

- Now, I would like to know if any member of the program committee of XXXX wrote papers on code smells. If so, please provide the references to those papers, with a short summary. You can find the list of member of the program committee at https://xxxxxx
- ChatGPT
  - Apologies, but as an AI language model, my responses are generated based on pre-existing training data and do not have real-time access to the internet or the ability to browse specific websites. Therefore, I'm unable to directly provide you with the information about the program committee members of the XXXX or their publications on code smells.
  - However, you can follow these steps to find the information you need:
    - […]
  - By following these steps, you should be able to find the relevant publications on code smells authored by the program committee members of XXXX.

- Thanks for the explanation. So, for instance, if I ask you to check if YYY (form the University yyyyy) wrote any paper dealing with code smells, you replay based uniquely on your training, without looking at any possible data that are available on the web?
- ChatGPT
  - That's correct. As an AI language model, my responses are generated based on the training data I've been provided, which includes a wide range of information up until September 2021. I don't have direct access to the internet, databases, or real-time information. Therefore, my responses are limited to the knowledge I have been trained on and cannot provide information on recent publications or specific details about individuals unless they are part of the existing training data.

# Focused requests

- Since the produced paper was clearly too short, I proceeded to prompt the GAI to produce additional text concerning specific sections of the SLR.

# Additional requests

- Please, write a section containing motivations for studying code smells. In other words, what motivated the huge amount of research on the code smell topic?
- Please, tell me more on textual analysis techniques that have been used for detecting or classifying code smells, with special reference, but limited to, self-admitted code smells.
- Self-admitted code smells hard to detect using automatic tools? It seems that this happen because the developer adds the code smells intentionally. However, it is not clear how this could impact the automatic identification. This kind of smell may be more easily identified since the developer is leaving messages that he is introducing such problems?

# Additional requests

- Could you give examples of the code smells mentioned above?
- Could you please give examples of self-admitted  code smells that are hard to detect using automatic tools?
- Which kind of context makes identifying self-admitted code mells harder for the automated tools?
- Concerning scientific papers, what are the advantages (if any) of a systematic literature review over a systematic mapping, especially with respect to characterizing the state-of-the-art?
- What are the reasons for preferring a SLR over a SMS?

# Results obtained

- The GAI fulfilled all the requests
- However, the provided answers had to be revised and adapted. Quite often, irrelevant or trivial considerations had to be discarded.

# About the primary studies

- I had to select them manually, querying a paper repository.

# The resulting paper

- A 4 pages and a half paper.
- Contents: reasonable, but shallow.

# The resulting paper

- Abstract & keyworks: OK
- Introduction: Acceptable. Could be better
- Methodology: Very poor (too short, too shallow)

# A Systematic Literature Review on Self-Admitted Code Smells

Anonymous Author(s)


## ABSTRACT

Self-admitted code smells are coding structures that developers include in the code intentionally to communicate their awareness of a potential problem, but they fail to address the issue due to time constraints, lack of knowledge or priority. Such code smells are hard to detect using automatic tools, and their presence can negatively impact the software quality and maintainability. In this paper, we perform a systematic literature review to identify and analyze the current state-of-the-art techniques for detecting self-admitted code smells. We searched the leading digital libraries and identified a total of 25 relevant studies published up to March 2023. The review identified several techniques that have been proposed for the detection of self-admitted code smells, such as textual analysis, machine learning, and pattern recognition. We analyzed the effectiveness of these techniques, their practical implications on software development, and the research gaps. Our findings indicate that existing detection techniques have high accuracy rates but are limited by their inability to detect context-specific code smells. Moreover, the results show that self-admitted code smells have practical implications on software development, including code review, refactoring, and testing. Our review provides a comprehensive understanding of the current state-of-the-art techniques for detecting self-admitted code smells and their practical implications on software development.



## KEYWORDS

Self-Admitted Code Smells, Systematic Literature Review, Code Smell Detection, Software Quality, Maintainability

**ACM Reference Format:**

Anonymous Author(s). 2018. A Systematic Literature Review on Self-Admitted Code Smells. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX


## 1 INTRODUCTION

Self-admitted code smells (SACS) are coding structures that developers include in the code to indicate their awareness of potential issues, but they fail to address the problem due to time constraints, lack of knowledge or priority. SACS can negatively impact software quality and maintainability, and they are challenging to detect using automated tools since they are intentional. Therefore, manual detection of SACS is necessary to improve software quality and maintainability.

The exploration of self-admitted code smells is an active area of researchy, which includes developing more sophisticated techniques for detecting and classifying self-admitted code smells, understanding their prevalence, analyzing their impact on software quality, and investigating how developers' awareness of code issues affects the overall development process.

In this paper, we perform a systematic literature review (SLR) to identify and analyze the current state-of-the-art techniques for detecting SACS.

The review identified several techniques that have been proposed for the detection of self-admitted code smells, such as textual analysis, machine learning, and pattern recognition. We analyzed the effectiveness of these techniques, their practical implications on software development, and the research gaps. Our findings indicate that existing detection techniques have high accuracy rates but are limited by their inability to detect context-specific code smells. Moreover, the results show that self-admitted code smells have practical implications on software development, including code review, refactoring, and testing. Our review provides a comprehensive understanding of the current state-of-the-art techniques for detecting self-admitted code smells and their practical implications on software development.

The rest of the paper is organized as follows. Section 2 described the method used to carry out the SLR. Section 3 illustrates the main findings of the survey. Section 4 reports about relevant related work. Finally, Section 5 draws some conclusions and sketches future research activities.

## 2 METHODOLOGY

We followed the guidelines provided by Kitchenham and Charters [19] for performing systematic literature reviews in software engineering. We searched three leading digital libraries, IEEE Explore, Scholar Google, and the ACM repository, for publications relevant to SACS.

We included publications written in English or Portuguese and published no later than March 2023. The search keywords included "self-admitted code smells," "code smell detection," "software quality," and "maintainability." The search yielded a total of 86 publications. After applying the inclusion and exclusion criteria, we identified 45 possibly relevant publications. After manually reviewing the abstracts and contents, we selected 25 relevant publications.

## 3 FINDINGS

In this section we report the main findings of the survey.

### 3.1 Yearly Distribution of Relevant Publications

Figure 1 shows the yearly distribution of relevant publications. The results a steady research interest that has been growing in 2022.



*Conference acronym 'XX, June 03–05, 2018, Woodstock, NY*



ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

# The resulting paper

- Figure 1 and Table 1: I had to do them
- Motivations for research on Code Smells: Acceptable, but rather trivial.
- Research areas: reasonable, except that references to primary studies are lacking.





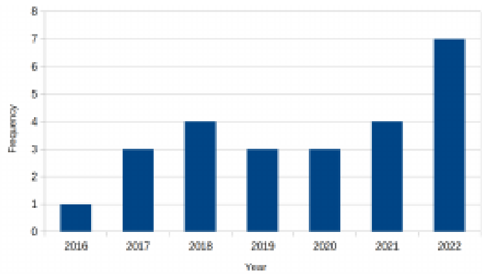


**Figure 1: Frequency of relevant publications per year**

### 3.2 Publication Venues

Table 1 shows the publication venues of the identified publications. The results indicate that most of the publications are published in software engineering conferences and journals.

**Table 1: Papers' frequency per venue.**

| Venue | N. | References |
|---|---|---|
| IEEE Transactions on SW Engineering | 3 | [7, 27, 28] |
| Empirical Software Engineering | 2 | [4, 15] |
| Information and Software Technology | 2 | [8, 18] |
| Other journals | 6 | [3, 5, 9, 21, 25, 26] |
| IEEE ICSME | 2 | [17, 29] |
| Other conferences | 6 | [2, 10, 12, 14, 30, 31] |
| Miscellaneous | 4 | [1, 6, 16, 23] |
| Total | 25 | |

In Table 1, the category "Miscellaneous" accounts for PhD Theses, Technical Reports, papers published in pre-print, etc. We included these papers because of their relevance, although not "properly" published.

### 3.3 Motivations for Research on Code Smells

According to the selected papers, the study of code smells has gained significant attention in the software engineering research community due to several compelling motivations. These motivations stem from the potential negative impacts of code smells on software quality, maintainability, and the overall development process. Understanding and addressing code smells have become crucial for software practitioners, researchers, and industry professionals alike. The motivations that the selected papers report to have driven the extensive research on code smells are described below.

**Quality Improvement**: Code smells are indicators of potential design and implementation issues in software systems. By studying and addressing code smells, researchers aim to improve the quality of software by identifying and rectifying suboptimal design choices, potential bugs, or inefficiencies. By detecting and eliminating code smells, software developers can enhance the maintainability, reusability, and readability of code, leading to more robust and reliable software systems.

**Maintenance and Evolution**: Code smells have been found to significantly impact the maintenance and evolution of software systems. Identifying and addressing code smells early in the development process can reduce the effort required for maintenance tasks, such as bug fixes, enhancements, and refactoring. By understanding the impact of code smells on software change-proneness, researchers seek to facilitate more efficient and cost-effective software maintenance and evolution.

**Software Metrics and Measurement**: Code smells provide quantifiable and measurable indicators of software quality. Researchers have explored the correlation between code smells and various software metrics, such as coupling, cohesion, and complexity. By studying code smells, researchers aim to develop reliable and accurate metrics for assessing software quality, enabling practitioners to monitor and manage code quality throughout the development lifecycle.

**Software Maintainability**: Code smells have a detrimental effect on software maintainability. They introduce complexity, increase the likelihood of introducing bugs during maintenance, and hinder the understandability of code. Researchers investigate the impact of code smells on software maintainability and propose techniques to detect, refactor, and prevent code smells, thereby improving the long-term maintainability of software systems.

**Developer Productivity and Collaboration**: Code smells can impact developer productivity and collaboration. They hinder the comprehension of code, increase the time required for bug fixes and feature implementation, and create obstacles in collaborative development scenarios. By studying code smells, researchers aim to provide developers with tools, techniques, and guidelines to detect and refactor code smells effectively, enhancing productivity and promoting collaboration within software development teams.

**Software Engineering Education**: Code smells serve as educational examples of common design and implementation issues. They are used to educate software engineering students and practitioners about good software design principles and best practices. By studying code smells, researchers contribute to the development of educational materials, case studies, and tools that aid in teaching and learning effective software engineering techniques.

These motivations have driven the extensive research on code smells, leading to a deeper understanding of their nature, impact, detection techniques, and practical implications on software development. By addressing code smells, researchers and practitioners aim to create high-quality, maintainable, and efficient software systems that meet user requirements and industry standards.

### 3.4 Research Areas

The selected papers address several research areas, as described below.

**Automated Code Smell Detection**: Developing more advanced and accurate automated techniques for code smell detection is an ongoing research area. This includes exploring machine learning approaches, such as deep learning and ensemble methods, to improve the precision and recall of code smell detection tools.

# The resulting paper

- SACS detections techniques: reasonable, except that references to primary studies are lacking.





**Contextual Code Smell Detection**: Code smells can be context-dependent, meaning their detection and severity can vary depending on the specific software system or programming language. Research is being conducted to develop techniques that consider the context, such as project-specific characteristics or domain knowledge, to enhance the accuracy of code smell detection and reduce false positives.

**Refactoring Assistance and Recommendations**: Providing intelligent refactoring assistance and recommendations based on detected code smells is an active area of research. This involves developing tools that not only identify code smells but also suggest appropriate refactoring techniques to address them effectively.

**Code Smell Evolution and Impact**: Understanding the evolution and impact of code smells over time is another area of interest. Research is being conducted to analyze how code smells propagate, accumulate, or diminish in software systems during maintenance and evolution, and how they impact other software quality attributes.

**Relationships between Code Smells**: Investigating the relationships between different code smells can provide insights into their co-occurrence patterns, dependencies, or synergies. Research is being conducted to understand how certain code smells are correlated or interact with each other, which can help in developing more holistic code smell detection and refactoring techniques.

**Integration with Development Tools**: Integrating code smell detection and refactoring tools directly into integrated development environments (IDEs) and software development workflows is an ongoing research area. The goal is to provide real-time feedback and assistance to developers, enabling them to identify and address code smells more efficiently during the coding process.

**Code Smells in Specific Contexts**: Exploring code smells in specific contexts, such as mobile applications, distributed systems, or machine learning code, is an emerging research area. Understanding how code smells manifest and impact different types of software systems can lead to tailored detection and refactoring techniques for specific domains.

**Code Smell Visualization and Explainability**: Visualizing code smells and their impact on software systems can aid in better understanding and decision-making. Research is being conducted to develop visualizations that highlight code smells, their relationships, and their consequences, as well as techniques to explain the detection results to developers.

**Empirical Studies and Benchmarking**: Conducting large-scale empirical studies and creating benchmark datasets for code smell detection and classification is crucial for evaluating the effectiveness of different techniques and comparing their performance. This helps in establishing best practices and identifying areas that require further improvement.

### 3.5 SACS Detection Techniques

The literature review identified several techniques that have been proposed for the detection of SACS, such as textual analysis, machine learning, and pattern recognition. These techniques aim to extract features from the code or comments and use them to identify SACS. Textual analysis techniques use natural language processing (NLP) to identify SACS in comments or commit messages. Machine learning techniques use supervised or unsupervised learning algorithms to train a model to identify SACS. Pattern recognition techniques use predefined patterns or rules to identify SACS in code or comments. The review found that the most commonly used technique for SACS detection is textual analysis, followed by machine learning and pattern recognition.

Textual analysis techniques that have been widely employed for detecting and classifying code smells, with special reference to self-admitted code smells are described below.

**Keyword-based Approaches**: Keyword-based approaches involve identifying specific keywords or phrases in textual artifacts that indicate the presence of code smells. Researchers define sets of keywords that are typically associated with code smells and use them as indicators for detection. For self-admitted code smells, specific keywords or phrases in code comments or commit messages can reveal developers' awareness of the code issues. By searching for such keywords, the presence of self-admitted code smells can be detected.

**Pattern Matching**: Pattern matching techniques involve searching for predefined patterns or regular expressions in textual artifacts to identify code smells. These patterns may capture common linguistic structures or conventions associated with code smells. For example, specific patterns can be used to identify code comments that explicitly admit to the presence of a code smell or use certain phrases that suggest the existence of a code issue.

**Machine Learning and Text Classification**: Machine learning techniques, such as supervised learning algorithms, can be employed to classify textual artifacts into categories related to code smells. Training datasets consisting of labeled examples (e.g., self-admitted code smells and non-self-admitted code smells) are used to train models that can automatically classify new instances. These models can learn the linguistic patterns, word frequencies, or semantic features associated with different types of code smells, including self-admitted ones.

**Topic Modeling**: Topic modeling techniques, such as Latent Dirichlet Allocation (LDA), can be applied to identify latent topics in textual artifacts related to code smells. By analyzing the distribution of topics in code comments or other textual sources, topic modeling can uncover discussions or mentions related to code smells, including self-admitted ones. This approach is particularly useful when the explicit keywords or patterns associated with self-admitted code smells are not consistently present.

**Sentiment Analysis**: Sentiment analysis techniques aim to determine the sentiment or emotional tone expressed in textual artifacts. In the context of code smells, sentiment analysis can be applied to identify code comments or discussions that express dissatisfaction, frustration, or acknowledgment of code issues. Sentiment analysis can provide valuable insights into the perception of self-admitted code smells and their potential impact on software quality.

These textual analysis techniques, whether used individually or in combination, contribute to the detection and classification of self-admitted code smells. By leveraging NLP and machine learning, these techniques enable researchers and practitioners to extract valuable information from textual artifacts and gain a deeper understanding of code quality issues within software projects.

# The resulting paper

- Effectiveness of SACS Detection Techniques: poor (too short, no data, no references)
- Practical implications: poor (too short, no original contribution, no references)
- Related work: minimal
- Conclusions: acceptable (also because one does not expects too much from Conclusions)



### 3.6 Effectiveness of SACS Detection Techniques

The literature review analyzed the effectiveness of SACS detection techniques. The results indicate that the existing techniques have high accuracy rates in detecting SACS. For instance, some studies reported accuracy rates of up to 91%. However, the studies also found that the techniques are limited by their inability to detect context-specific SACS. Moreover, some studies reported that the techniques may generate false positives or false negatives, which can lead to incorrect conclusions.

### 3.7 Practical Implications on Software Development

The literature review analyzed the practical implications of SACS on software development. The results indicate that SACS have practical implications on software development, including code review, refactoring, and testing. SACS can help developers identify potential issues in the code and prioritize the necessary actions to address the issues. Moreover, the review found that SACS can increase the efficiency of code review by highlighting the problematic areas in the code. Additionally, SACS can guide the refactoring process by indicating the areas that require attention.

## 4 RELATED WORK

Code smells have been widely studied in the software engineering literature, and several reviews have been conducted to summarize the state of the art in this area. This section provides a selection of key references in the field of code smells.

Marinescu [22] provides a comprehensive overview of code smells and their impact on software quality. The paper also proposes metrics-based rules for detecting design flaws, including code smells, in object-oriented software.

Fowler introduces the concept of refactoring as a technique to improve code quality, maintainability, and design [11]. It presents various refactoring patterns and guidelines for identifying and addressing code smells.

Lanza and Marinescu explore the application of software metrics to characterize and evaluate the design quality of object-oriented systems [20]. They discusses various metrics related to code smells and their practical use in assessing and improving software design.

Research on code smells has been favored by initiatives like the one by Gousios and Spinellis [13], who prrovided the GHTorrent dataset, which includes comprehensive data from GitHub repositories. The dataset enables research on code smells, among other software engineering topics, by offering access to a large-scale and diverse collection of open-source projects.

Palomba et al. investigated developers' perception of bad code smells [24]. The paper presents a study involving developers' evaluation of code examples with and without code smells, providing insights into the subjective perception of code smells and their impact on maintainability.

## 5 CONCLUSION

The literature review provides a comprehensive view of the current state-of-the-art techniques for detecting SACS and their practical implications on software development.

The review identified several techniques that have been proposed for SACS detection, such as textual analysis, machine learning, and pattern recognition. Similarly, the review identified several proposals concerning the analysis of SACS detection effectiveness, and their practical implications on software development.

The results indicate that the existing techniques have satisfactory accuracy rates in detecting SACS but are limited by their inability to detect context-specific SACS.

The survey also reports practical benefiots for software development, specifically concerning code review, refactoring, and testing.

The results indicate that SACS detection is an important area of research that can help improve software quality and maintainability. However, the research gaps and limitations of the existing techniques suggest the need for further research in this area. In particular, it appears that a much needed improvement concerns the ability to relate SACS to "environmental" characteristics, e.g., application-specific issues, development propcess issues, people issues, etc.

## REFERENCES

[1] Yaqoob Salman Al-Slais. 2021. Towards a comprehensive self-admitted technical debt extraction technique from source code comments. (2021).
[2] Eman AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. IEEE, 51–58.
[3] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, and Ali Ouni. 2021. Toward the automatic classification of self-affirmed refactoring. *Journal of Systems and Software* 171 (2021), 110821.
[4] Hideaki Azuma, Shinsuke Matsumoto, Yasutaka Kamei, and Shinji Kusumoto. 2022. An empirical study on self-admitted technical debt in Dockerfiles. *Empirical Software Engineering* 27, 2 (2022), 49.
[5] Xin Chen, Dongjin Yu, Xulin Fan, Lin Wang, and Jie Chen. 2021. Multiclass classification for self-admitted technical debt based on XGBoost. *IEEE Transactions on Reliability* 71, 3 (2021), 1309–1324.
[6] Everton da S Maldonado. 2016. *Identifying Self-Admitted Technical Debt*. Ph. D. Dissertation. Concordia University.
[7] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1044–1062.
[8] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, Marcos Kalinowski, and Rodrigo Oliveira Spínola. 2020. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology* 121 (2020), 106270.
[9] Bruno Santos de Lima, Rogerio Eduardo Garcia, and Danilo Medeiros Eler. 2022. Toward prioritization of self-admitted technical debt: an approach to support decision to payment. *Software Quality Journal* 30, 3 (2022), 729–755.
[10] Zhang Di, Bing Li, Zengyang Li, and Peng Liang. 2018. A preliminary investigation of self-admitted refactorings in open source software. In *International Conferences on Software Engineering and Knowledge Engineering*. KSI Research Inc. and Knowledge Systems Institute Graduate School.
[11] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
[12] Felipe Gustavo de S Gomes, Thiago Souto Mendes, Rodrigo O Spínola, Manoel Mendonça, and Mário Farias. 2019. Uma análise da relação entre code smells e dívida técnica auto-admitida. In *Anais do VII Workshop de Visualização, Evolução e Manutenção de Software*. SBC, 25–32.
[13] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: GitHub's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 12–21.
[14] Xiaofeng Han, Amjed Tahir, Peng Liang, Steve Counsell, and Yajing Luo. 2021. Understanding code smell detection via code review: A study of the openstack community. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 323–334.
[15] Azuma Hideaki, Matsumoto Shinsuke, Kamei Yasutaka, and Kusumoto Shinji. 2022. An empirical study on self-admitted technical debt in Dockerfiles. *Empirical Software Engineering* 27, 2 (2022).
[16] Alina Hrynko. 2020. Source code quality in connection to self-admitted technical debt.

# The resulting paper

- References: I had to write them!



[17] Martina Iammarino, Fiorella Zampetti, Lerina Aversano, and Massimiliano Di Penta. 2019. Self-admitted technical debt removal and refactoring actions: Co-occurrence or more?. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 186–190.

[18] Yutaro Kashiwa, Ryoma Nishikawa, Yasutaka Kamei, Masanari Kondo, Emad Shihab, Ryosuke Sato, and Naoyasu Ubayashi. 2022. An empirical study on self-admitted technical debt in modern code review. *Information and Software Technology* 146 (2022), 106855.

[19] B.A. Kitchenham and S. Charters. 2007. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE-2007-01. Keele University and Durham University.

[20] Michele Lanza and Radu Marinescu. 2007. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media.

[21] Hui Li, Yang Qu, Yong Liu, Rong Chen, Jun Ai, and Shikai Guo. 2022. Self-admitted technical debt detection by learning its comprehensive semantics via graph neural networks. *Software: Practice and Experience* 52, 10 (2022), 2152–2176.

[22] Radu Marinescu. 2004. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 350–359.

[23] Cédric Noiseux. 2017. *Recommending When Design Technical Debt Should Be Self-admitted*. Ecole Polytechnique, Montreal (Canada).

[24] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. 2014. Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 101–110.

[25] Luka Pavlič, Tilen Hliš, Marjan Heričko, and Tina Beranič. 2022. The Gap between the Admitted and the Measured Technical Debt: An Empirical Study. *Applied Sciences* 12, 15 (2022), 7482.

[26] Rishab Sharma, Ramin Shahbazi, Fatemeh H Fard, Zadia Codabux, and Melina Vidoni. 2022. Self-admitted technical debt in R: detection and causes. *Automated Software Engineering* 29, 2 (2022), 53.

[27] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1211–1229.

[28] Meng YAN, Xin XIA, Emad SHIHAB, David LO, Jianwei YIN, and Xiaohu YANG. 2018. Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering* (2018).

[29] Fiorella Zampetti, Cedric Noiseux, Giuliano Antoniol, Foutse Khomh, and Massimiliano Di Penta. 2017. Recommending when design technical debt should be self-admitted. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 216–226.

[30] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2018. Was self-admitted technical debt removal a real removal? an in-depth perspective. In *Proceedings of the 15th international conference on mining software repositories*. 526–536.

[31] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2020. Automatically learning patterns for self-admitted technical debt removal. In *2020 IEEE 27th International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 355–366.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

# The final result: reactions from the PC

- No comment mentioned the usage of GAI.
  - That is, the reviewers did not suspect that the paper had largely be produced by GAI.
- The paper was rejected (with reason)
  - Essentially, because it was too short and shallow
  - Nothing wrong was found, though

# Questions form the PC

- Q1.: Why are SACs hard to detect using automatic tools? You claim that this happen because the developer adds the code smells intentionally. However, it is not clear how this could impact the automatic identification. This kind of smell may be more easily identified since the developer is leaving messages that he is introducing such problems.
- Q2.: Could you give examples of the code smells explored in your study?
- Q3.: Which kind of context is harder for the automated tools to identify the SACs?
- 1) Did you know there is no short paper in the research track?
- 2) Why did you not produce a 10-page paper?
- 3) Why do you characterize your paper as an SLR and not systematic mapping?
- Where is your detailed methodology section?

# On the required effort

- Effort was not negligible
  - Preparing questions for GAI requires some thinking
  - Understanding how GAI works, what it can and what it cannot do, etc. required some time
  - Some tasks were not at all supported by GAI
    - Retrieving references
    - Deriving data about venues and publication times
    - Preparing figures and tables
    - Making the text homogeneous and consequential
    - Formatting
- For producing a paper that had very little chances of being accepted

# Apologies

- I apologize for producing a paper and having it reviewed, even though I was not going to publish it eventually.
- It was for a good cause (this talk), anyway.

# Second attempt: using chatGPT 4

- I used ChatGPT 4 (paying) to address the biggest issues with the previous paper.
- Findings:
  1. Chat GPT 4 is less assertive. It tends to provide suggestions, rather than ready to use results.
     - This was OK for the methodology: the suggestions on how to carry out the survey became a description  of how it was carried out.
  2. Chat GPT 4 is no better with references: it invented them exactly as ChatGPT 3.5
  3. Like ChatGPT 3.5, also ChatGPT 4 is noty able to retrieve information from the internet.

# Access to papers

- The two versions of the paper are available at:
  - https://drive.google.com/drive/folders/1aaVMDjnnVAFBTpg7s9hgF2RsjpanNZhA?usp=sharing
  - Or bit.ly/lavazza-ICSEA23

# Comments and questions

- Who was the author of the paper?
  - GAI contributed a lot, but its product was definitely not an acceptable result.
- What if the paper had been accepted?
  - The resulting paper is not worse than several published papers I read…
- Would it have been ethical to publish it?