Tracing Feature Tests to Textual Requirements

Mahima Dahiya
University of Cincinnati
Cincinnati, OH, USA
dahiyama@mail.uc.edu

Mark Li
Avon High School
Avon, CT, USA
88markli@gmail.com

Glen Horton, Thomas Scherz, Nan Niu
University of Cincinnati
Cincinnati, OH, USA
{hortongn, scherztc}@ucmail.uc.edu, nan.niu@uc.edu

***Abstract*—Software features deliver values to the end users, and thus their qualities shall be assured. While the mainstream quality assurance technique is software testing, its adequacy can only be assessed by tracing the testing artifacts to system requirements. In this paper, we report an in-depth case study aiming to trace all the feature tests of a Web application to textual requirements. To that end, we experiment two automated trace recovery methods: vector space model and transformer-based semantic embedding. The tracing results, unfortunately, are not satisfactory. We then explore the use of large language models, OpenAI's ChatGPT in particular. We engage the original developers into evaluating the tracing and ChatGPT-prompting results. Our study reveals the promises of exploiting large language models to assist in software tracing activities.**

***Index Terms*—requirements traceability, software testing, trace recovery, large language models**

I. Introduction

Traceability—the ability to interrelate any uniquely identifiable software engineering artifact to one other [1]—has long been recognized as a desired quality attribute of a well-engineered software system. Traceability is so important that, without it, tasks like change impact analysis and system-level test coverage analysis could not be undertaken [2].

To assess the adequacy of system-level tests, one shall trace these tests to software requirements. In this paper, we thus focus on the tests written to verify the correct behaviors of features as the tracing sources, where a feature is an increment of functionality that delivers user-visible values. Our work's tracing targets are textual requirements because natural language remains the most dominant form for expressing and conveying stakeholders' needs and desires [3].

Manual tracing is tedious, time-consuming, and error-prone. Automated methods commonly rely on the textual and semantic similarities between the software artifacts to determine whether a trace link should be established. Extensive empirical studies on automated tracing methods, such as vector space model and latent semantic indexing, indicate that almost the same traceability information is captured [4]. In most cases, a recall of 90% is achievable at precision levels of 5–30%, where recall measures coverage and is defined as the percentage of correct links that are established, and precision measures accuracy and is defined as the percentage of established links that are correct [5]–[7].

An underlying cause for the low accuracy levels is rooted in the concept assignment problem [8], i.e., people use their terminologies differently in software development [9]. The problem is especially challenging when tracing feature tests to textual requirements—two artifact types often separated from each other distinctively in the software engineering life cycle.

In this paper, we report an in-depth case study of tracing all the feature tests of a real-world software system. In total, 67 tests are used to help assure the qualities of the features of a production Web application written in Ruby on Rails. We first apply two automated tracing methods, namely vector space model [5] and transformer-based semantic embedding [10], to determine the top-matched requirement for each test. The requirements are taken from the user stories of the software project. In total, we extract 146 textual requirements. The tracing results, to our surprise, are unsatisfactory. For example, multiple tests are traced to the same requirement with either method, and many requirements are left with no test at all. A main reason is the project's use of the early-phase requirements [11], [12] for elicitation and design, rather than for verification and validation.

These results motivate us to explore the use of large language models, in particular OpenAI's ChatGPT, to generate the description of the feature that a test is intended to verify. In this way, the ChatGPT results may align better with the testing artifacts. To that end, we experiment two ways to prompt ChatGPT: a straightforward baseline and a pattern-enriched variant by incorporating White *et al.*'s recent work [13]. We then present the traced and the ChatGPT-generated results to the original developers, asking them to assess the accuracy and appropriateness of these results.

This paper makes two key contributions. First, we show that the state-of-the-art methods do not necessarily produce satisfactory tracing results, challenging the recovery mode of traceability. Second, we exploit large language models to instrument traceability in a generative way, and point out potential pitfalls of prompt patterns [13] applied in traceability. In what follows, we present our case study design in Section II. We then analyze the study results in Section III, discuss related work in Section IV, and conclude the paper in Section V.

II. Design of An Exploratory Case Study

A case study is an empirical inquiry that investigates a contemporary phenomenon within its real-life context [16]. Wohlin [17] emphasizes that a case study shall draw on multiple sources of evidence. We use an exploratory case study [16] as the basis for our research design to understand the tracing of feature tests to textual requirements.

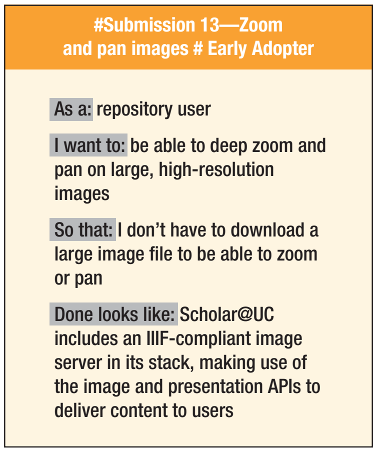
```
  1  # frozen_string_literal: true
...  ...
 56    describe 'updating embargoed object' do
...  ...
 79      it 'can be updated with a valid date' js: true do
 80        visit "/concern/generic_work/#{work.id}"
 81
 82        click_link 'Edit'
 83        click_link 'Embargo Management Page'
 84
 85        expect(page).to have_content('This Generic Work is under embargo.')
 86        expect(page).to have_xpath("//input[@name='generic_work[embargo_release_date]' and @value='#{future_date.to_datetime.iso8601}']")
 87
 88        fill_in 'until', with: later_future_date.to_s
 89
 90        click_button 'Update Embargo'
 91        expect(page).to have_content(later_future_date.to_date.to_formatted_s(:standard))
 92      end
 93
 94      it 'cannot be updated with an invalid date' do
 95        visit "/concern/generic_work/#{work.id}"
 96
 97        click_link 'Edit'
 98        click_link 'Embargo Management Page'
 99
100        expect(page).to have_content('This Generic Work is under embargo.')
101        expect(page).to have_xpath("//input[@name='generic_work[embargo_release_date]' and @value='#{future_date.to_datetime.iso8601}']")
102
103        fill_in 'until', with: invalid_future_date.to_s
104
105        click_button 'Update Embargo'
106        expect(page).to have_content('Release date specified does not match permission template release requirements for selected AdminSet.')
107      end
108    end
109  end
```

Fig. 1. Snippet of a feature test organized in an .rb file [14]; this artifact has a total of 109 lines of test code, among which the "expect" keyword (e.g., in lines 85, 86, and 91) defines the test oracle [15].

The subject system of our case study is a Web application: Scholar@UC [18]. The application has been deployed to serve a higher-educational institution's user community for almost a decade. The Web application is written in Ruby on Rails. Accordingly, the development team creates the feature tests via RSpec [19], which supports behavior-driven development and is the most frequently used testing library for Ruby in production applications [20]. In total, Scholar@UC has 67 feature tests, each of which is organized in an .rb file. Fig. 1 illustrates one feature test [14]. All the testing artifacts are publicly available [21].

While the tests are readily available, we extract the requirements from Scholar@UC's user stories [22]. Fig. 2 shows a sample user story about the feature supporting images compliant with the International Image Interoperability Framework (IIIF). We use only the 'Done looks like' part to represent the functional aspects of the software system, and ignore who ("As a") intends to do what ("I want to") in order to achieve which goal ("So that"). Thus, the textual requirement that we derive from Fig. 2 is: "The system includes an IIIF-compliant image server in its stack, making use of the image and presentation APIs to deliver content to users." Note that we replace "Scholar@UC" with "the system". In total, we construct 146 textual requirements by covering all the 146 user stories from the Scholar@UC project [22].

To explore the tracing of Scholar@UC's feature tests to textual requirements, we set out to answer two research questions (RQs).



#Submission 13—Zoom and pan images # Early Adopter

As a: repository user

I want to: be able to deep zoom and pan on large, high-resolution images

So that: I don't have to download a large image file to be able to zoom or pan

Done looks like: Scholar@UC includes an IIIF-compliant image server in its stack, making use of the image and presentation APIs to deliver content to users

Fig. 2. A user story from the Scholar@UC project [12], [22].

RQ$_1$: How do the automated tracing methods perform in linking Scholar@UC's feature tests to the existing textual requirements?

Because each of the 67 .rb files (cf. Fig. 1) is meant to test a single feature of Scholar@UC, we are interested in the top-matched linking result. For each .rb file, we apply two automated trace recovery methods to find the top-matched requirement from the 146 candidates. One method is the vector space model, which represents a software artifact in a high-dimensional space by treating the terms contained in the artifact as dimensions. Our Python implementation uses sklearn's cosine_sim to compute the textual similarity between two artifacts after transforming them with tfidf_matrix. We denote this method as Cosine.

While Cosine considers the text corpus formed only by the software artifacts, we also experiment with a pre-trained deep learning model: RoBERTa (a robustly optimized BERT pre-training approach) [23]. RoBERTa builds on the now ubiquitous transformer architecture and overcomes the undertrained BERT with dynamic masking, full-sentences, larger byte-pair encoding, and other optimization mechanisms. Our Python implementation uses PyTorch's SentenceTransformer model. The RoBERTa method thus captures semantic similarity with the SentenceTransformer's pretraining data that go beyond the software artifacts' corpus.

RQ$_2$: Would ChatGPT trace Scholar@UC's feature tests in a generative way?

As we show in the next section that neither Cosine nor RoBERTa recovers the trace links well, we investigate the potential of OpenAI's ChatGPT—a prominent large language model—in generating the textual description of the feature tested by an .rb file. We exploit gpt-3.5-turbo [24] and experiment two prompts. The first prompt, which we refer to as "baseline", invokes ChatGPT with the request: "Can you please suggest a feature description for the above code?"

The second prompt builds on the patterns codified by White *et al.* [13]. Specifically, we incorporate the "Meta Language Creation" pattern by pointing out the test oracle defined by the "expect" keyword (cf. Fig. 1). In this way, we want ChatGPT to understand and pay attention to the important input semantics. The prompt, which we call "pattern-enriched", is: "Please suggest a feature description for the above code in a complete sentence and when you see the keyword 'expect', it defines the test oracle. Try to incorporate the test oracle or test oracles naturally in your response. Try to keep the answer in no more than 20 words."

To evaluate the qualities of the tracing results from Cosine and RoBERTa, as well as the ChatGPT results from the baseline and pattern-enriched prompting, we engage two Scholar@UC developers in a survey. We randomly choose 7 .rb test files, accounting for 10% of all the feature tests, and ask the developers to determine which one from the (i) Cosine, (ii) RoBERTa, (iii) baseline prompting, and (iv) pattern-enriched prompting results best matches the .rb file. If none of the four choices are deemed to be viable, the developers can provide their own answers. It is worth noting that, by empirically inquiring Scholar@UC's project repositories and its developers, our case study draws on multiple sources [17] to gain insights into the tracing practices of a real-world software system.

III. Results and Analysis

A. Tracing Results

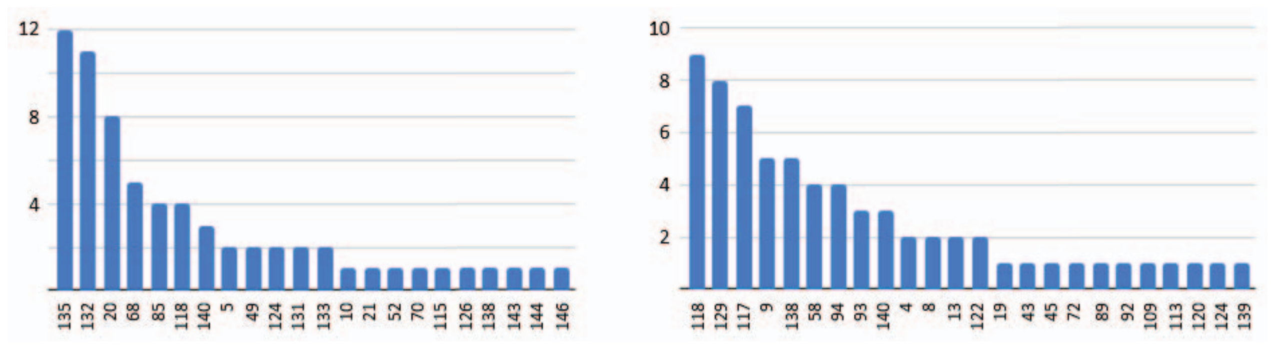
The Cosine and RoBERTa tracing results are summarized in Fig. 3. Both frequency charts show that multiple tests are traced to the same requirement. In Fig. 3-a, for example, Cosine traces 12 different .rb files to the same requirement, and in Fig. 3-b, RoBERTa maps 9 test files to one tracing target. These results are not satisfactory, because we would expect little to no overlap of different testing artifacts when it comes to identifying the features that they are testing. In case of a high degree of overlap, such as Cosine's 12:1 and RoBERTa's 9:1, testing effort would have become redundant, and the development team of Scholar@UC would have reduced the redundancy, e.g., by consolidating or refactoring the test code.

Table I provides the descriptions of the top-traced requirements, along with the number of times Cosine and RoBERTa trace the test files to them. For example, requirement 118, "When the repository users browse for a work to attach from their computer, they are able to change the file name before submitting the work for the first time", is the top trace for 4 .rb files with Cosines, and 9 .rb files with RoBERTa. It can be seen from Table I that Cosine and RoBERTa differ in their tracing results. For instance, requirement 135 is Cosine's top-traced result for 12 tests, but RoBERTa maps this requirement to none of the 67 tests. In fact, only one test—concern_show_spec.rb—is traced to the same requirement (ID: 138) by both Cosine and RoBERTa, suggesting the very different behaviors of the two tracing methods. We speculate that the difference can be attributed to Cosine's artifact-only corpus and RoBERTa's broadened pretraining corpus. Nevertheless, which tracing method is better, or given the unsatisfactorily high overlap degrees, could both methods be having subpar performances? Our developer survey sheds light on these questions, and we analyze the survey results in the next sub-section.

From the requirements' perspective, the tracing results are also less ideal. Although the 67 .rb test files cannot cover all the 146 requirements theoretically, Cosine's and RoBERTa's tracing results contain only 22 and 24 requirements respectively. A closer look at the requirements reveals that Scholar@UC's user stories mainly help elicitation and scoping the design in software development [11]. The "# Early Adopter" tagged in the user story of Fig. 2 signals this aim. Here, the Scholar@UC project engages enthusiastic users in understanding their needs and preferences [12], [25]. The actual functionality that is implemented and tested may or may not align well with such elicitation-focused requirements. Thus, misaligned or outdated artifacts negatively impact the performances of trace recovery methods like Cosine and RoBERTa. We next analyze the results of instrumenting traceability in a generative fashion.

B. Prompting and Survey Results

Given that the trace recovery results are less satisfactory, we prompt ChatGPT to generate the textual description of what feature an .rb file is testing. As discussed in Section II, we experiment two prompts: baseline that asks ChatGPT straightforwardly, and pattern-enriched that instructs ChatGPT to also pay attention to the test oracle specified with the "expect" keyword. To answer **RQ**$_2$, we randomly sampled



(a) Cosine **(b) RoBERTa**

Fig. 3. Tracing results of Scholar@UC's 67 feature tests: x-axis represents the ID of the requirements, each of which is the top-matched trace of at least one feature test, and y-axis represents the number of feature tests tracing to the requirement.

TABLE I
TOP-TRACED REQUIREMENTS: THE TWO RIGHTMOST COLUMNS SHOW THE NUMBER OF TIMES A SPECIFIC REQUIREMENT IS THE RESULT OF THE TOP-MATCHED TRACE BY COSINE AND ROBERTA RESPECTIVELY

ID	Requirements Description	Cosine	RoBERTa
118	When the repository users browse for a work to attach from their computer, they are able to change the file name before submitting the work for the first time.	4	9
135	When the repository user logs in to the system, the user receives a notice that the TOU or distribution license have changed, or the user receives an automatically generated email.	12	0
132	When a repository user is ready to upload, the user can enter a name of the journal that previously entered the work, and it can tell the user whether the user may proceed or not.	11	0
20	Users on a collection page can select which files they want to download and bulk download them from the collection page.	8	0
129	There will be a clearer interface for expanding the "Show Additional Description" section, reusing some of the UI elements from elsewhere in the design (e.g., the "Add+" button).	0	8
117	After the repository user selects a file from the local system, the file name is automatically populated in the title field. The user may edit this title for clarity.	0	7

TABLE II
SEVEN RANDOMLY SAMPLED FEATURE TESTS: THE FULL PATH IS "https : //github.com/uclibs/ucrate/blob/develop/spec/features/" + Name, AND Size IS GIVEN IN THE NUMBER OF LINES OF CODE

Name	Size	Keyword
doi_request_spec.rb	13	DOI
hyrax/search_spec.rb	69	Search
display_admin_dashboard_spec.rb	34	Dashboard
create_document_spec.rb	147	Creation
capatability_list_route_spec.rb	10	Capability
rss_feed_spec.rb	29	RSS
hyrax/embargo_spec.rb	109	Embargo

7 tests, as shown in Table II. As indicated by the .rb files' names, these tests focus on different functional aspects of the software system. We use an individual keyword related to functionality to refer to each test in the rest of this section. These keywords, such as "DOI", "Search", and "Dashboard", are listed in Table II. Additionally, Table II shows that the sizes of our sampled tests are from 10 to 147 lines of code, implying a diverse range over the population (mean=97.8 and median=72 of all the 67 .rb files).

The tracing and prompting results of the seven tests are shown in Table III. We make a couple of observations. First, the Cosine tracing results exhibit a high degree of overlap; in particular, 'Search' and 'Capability' are traced to the same requirement, and so are 'RSS' and 'Embargo'. Second, the pattern-enriched prompting invariably includes 'expect' in the results. Recall that our prompt instructs ChatGPT to *naturally* incorporate the input semantics into the results; however, it seems that ChatGPT can do this only syntactically.

To assess the qualities of the tracing and prompting results, we surveyed two Scholar@UC developers. Specifically, we designed an online survey via Google Forms, randomly ordering the four options for each test file. For each file, our survey question was: "What is the .rb file testing?" We only shared the textual results and intentionally hid the methods from which the results were generated. The two developers spent several days on the survey individually, and then jointly filled out the Google Forms.

The survey results are summarized in Table IV. The developers chose none of Cosine or RoBERTa results, confirming that the requirements extracted from user stories might not align well with the actual implementation and verification. Surprisingly, the pattern-enriched prompting was not always preferred by the developers. One developer commented that, "The options that mentioned 'expect statements' and 'test oracles' started with good descriptions of the feature, but ended

TABLE III
TRACING AND PROMPTING RESULTS OF SEVEN TESTS

Keyword	Tracing Results		ChatGPT-Prompting Results	
	Cosine	RoBERTa	Baseline	Pattern-Enriched
DOI	When logged in, the repository manager is able to request a report of repository interactions for a given period of time.	The submission interface includes mechanism for generating DOIs, which are displayed on the pubic interface for the item.	Tests the DOI request functionality for various types of content in the system.	The feature tests DOI request behavior for various work types; 'expect' statements define test oracles, verifying expected system responses.
Search	Users on a collection page can select which files they want to download and bulk download them from the collection page.	When a repository user searches for a compound term, punctuation and other related forms are treated as white-space or stemmed as appropriate.	Validates search functionality for public users, covering gallery view, search options display, and browse button visibility.	The feature tests searching functionality as a public user, verifying expected page content and elements using 'expect' statements as test oracles.
Dashboard	If repository user's nomination is selected for further web archiving, the repository user can return to the repository to see the archived version of the web page.	After the repository user selects a file from the local system, the file name is automatically populated in the title field. The user may edit this title for clarity.	Tests the dashboard view and functionality for an admin user account.	The feature tests the admin dashboard, confirming expected elements and links using 'expect' statements as test oracles for admin user views.
Creation	When the repository user logs in to the system, the user receives a notice that the TOU or distribution license have changed, or the user receives an automatically generated email.	When the repository user attaches subsequent files to the work, the user can select different rights statements for those files.	Verifies the creation and metadata display of a Document by a logged-in user.	This feature tests the document creation process, validating form inputs and expected behavior using 'expect' statements as test oracles.
Capability	Users on a collection page can select which files they want to download and bulk download them from the collection page.	The SHERPA ROMEO link opens in a new window or the repository submitter should be warned not to use the back button.	Ensures proper redirection to error page when visiting capability list path.	This feature tests the redirection of users to the sanitize error page when visiting the capability list path, using 'expect' statements as test oracles to verify the presence of the 'Page Not Found' content.
RSS	When the link is displayed on the 'show page' for a work, the title and description will also be displayed.	There will be a clearer interface for expanding the "Show Additional Description" section, reusing some of the UI elements from elsewhere in the design (e.g., the "Add+" button).	Verifies the RSS feed functionality, including item count and exclusions.	This feature tests the RSS feed's functionality by ensuring it returns the specified number of items, excludes private works, and excludes collections, using 'expect' statements as test oracles to verify the expected content presence or absence.
Embargo	When the link is displayed on the 'show page' for a work, the title and description will also be displayed.	The repository user could enter the journal name into a field and the system would detect when the user's article is published, and remove the embargo from the deposited content.	Tests embargo creation, display, and update with valid and invalid release dates.	This feature tests the creation and update of embargoed objects, ensuring proper functionality by verifying expected content presence or absence using 'expect' statements as test oracles.

TABLE IV
SURVEY RESULTS FROM SOFTWARE DEVELOPERS

Keyword	Cosine	RoBERTa	Baseline	Pattern
DOI			●	
Search			●	
Dashboard			●	
Creation				●
Capability			●	
RSS				●
Embargo*				

*The developers chose 'Other:' and entered: "Tests embargo creation and update with valid and invalid release dates."

with details describing the test as well and was often awkward when compared to the other options." Thus, readability is important for ChatGPT's result generation. The other developer remarked that, "The two 'expect' versions that we thought did a better job, because either it got the specifics correct (e.g., items of a form) or better described an important feature of the functions, like exclusions." Clearly, technical depth and accuracy are of crucial importance for ChatGPT to support software engineering tasks. Finally, rather than choosing any of the four options, the developers responded, "Tests embargo creation and update with valid and invalid release dates", as the answer to the Embargo test. This indicates areas for improvement in test-to-requirement tracing and prompting.

C. Threats to Validity

The construct validity of our exploratory case study can be affected by our unit of analysis at the test file level. In Fig. 1, for example, the "`do ... end`" block from line 79 to line 92 defines one test case, and so does the block from line 94 to line

107. The former tests an embargo object's update with a valid date, whereas the latter with an invalid one [26]. Although traceability can be established at the test case level, we believe the encapsulation of multiple test cases in one file represents the testing of a coherent feature in an adequate manner.

A threat to internal validity is our prompting of ChatGPT for creating the tracing targets. Although Table IV favors the results from the baseline prompting, it remains somewhat unclear whether the baseline results are descriptions of user-visible features, or summaries of the test files. As discussed earlier, the pattern-enriched prompting results in awkward readability oftentimes, leaving room for future research to better work with ChatGPT to improve its task understandability.

Threats to the external validity include our investigations of only one software system, though our study has covered all of the system's feature tests. In addition, we use only gpt-3.5-turbo, and therefore the prompting results may differ with other large language models.

IV. Related Work

Software traceability has been studied extensively in the literature, ranging from motivations and benefits to trace link recovery and maintenance [1]. Many researchers have applied information retrieval techniques to automatically link different software artifacts. Guo *et al.* [27] were among the first to apply deep learning in automated traceability. We experiment not only a classic information retrieval method (namely Cosine), but also a deep learning mechanism (namely RoBERTa).

The emergence of large language models attracts immediate attention among requirements engineering researchers. For example, Fantechi *et al.* [28] used ChatGPT to detect inconsistencies in requirements documents, and Rodriguez *et al.* [29] realized that the explanations provided by a large language model were not only detailed but also useful for refining the prompts to advance automated traceability. Our work differs from [29] in that we invoke ChatGPT to generate a tracing target, rather than generating link explanations.

V. Conclusions

We have shown in our case study that merely recovering trace link can be insufficient due to the distinct purposes served by the various artifacts. Furthermore, we show that ChatGPT can potentially generate the tracing targets, offering a new way to address the traceability challenge. Our future work includes experimenting more prompting patterns, performing theoretical replications [30], and using traceability to improve test coverage [31].

References

[1] J. Cleland-Huang *et al.*, "Software traceability: Trends and future directions," in *FOSE*, Hyderabad, India, May-June 2014, pp. 55–69.

[2] J. Hayes *et al.*, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Trans. Software Eng.*, vol. 32, no. 1, pp. 4–19, January 2006.

[3] M. Kassab *et al.*, "State of practice in requirements engineering: Contemporary data," *Innovations in Systems and Software Engineering*, vol. 10, no. 4, pp. 235–241, December 2014.

[4] R. Oliveto *et al.*, "On the equivalence of information retrieval methods for automated traceability link recovery," in *ICPC*, Braga, Portugal, June 2010, pp. 68–71.

[5] G. Antoniol *et al.*, "Recovering traceability links between code and documentation," *IEEE Trans. Software Eng.*, vol. 28, no. 10, pp. 970–983, October 2002.

[6] A. Mahmoud and N. Niu, "Source code indexing for automated tracing," in *TEFSE*, Honolulu, HI, USA, May 2011, pp. 3–9.

[7] W. Wang *et al.*, "Enhancing automated requirements traceability by resolving polysemy," in *RE*, Banff, Canada, August 2018, pp. 40–51.

[8] T. J. Biggerstaff *et al.*, "Program understanding and the concept assignment problem," *Comm. ACM*, vol. 37, no. 5, pp. 72–82, May 1994.

[9] N. Niu and S. Easterbrook, "So, you think you know others' goals? A repertory grid study," *IEEE Software*, vol. 24, no. 2, pp. 53–61, March/April 2007.

[10] J. Lin *et al.*, "Enhancing automated software traceability by transfer learning from open-world data," *CoRR*, July 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2207.01084

[11] E. Yu, "Towards modeling and reasoning support for early-phase requirements engineering," in *RE*, Annapolis, MD, USA, January 1997, pp. 226–235.

[12] N. Niu *et al.*, "Requirements engineering and continuous deployment," *IEEE Software*, vol. 35, no. 2, pp. 86–90, March/April 2018.

[13] J. White *et al.*, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," *CoRR*, February 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.11382

[14] University of Cincinnati, "Embargo Feature Test of Scholar@UC," Last accessed: June 6, 2024. [Online]. Available: https://github.com/uclibs/ucrate/blob/develop/spec/features/hyrax/embargo_spec.rb

[15] E. T. Barr *et al.*, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, May 2015.

[16] R. K. Yin, *Case Study Research: Design and Methods*. SAGE Publications, 2003.

[17] C. Wohlin, "Case study research in software engineering—it is a case, and it is a study, but is it a case study?" *Information & Software Technology*, vol. 133, pp. 106 514:1–106 514:3, May 2021.

[18] University of Cincinnati, "Scholar@UC," Last accessed: June 6, 2024. [Online]. Available: https://scholar.uc.edu/

[19] J. Rowe *et al.*, "RSpec: Behaviour Driven Development for Ruby," Last accessed: June 6, 2024. [Online]. Available: https://rspec.info/

[20] M. Anastasov, "Getting started with RSpec," Last accessed: June 6, 2024. [Online]. Available: https://semaphoreci.com/community/tutorials/getting-started-with-rspec

[21] University of Cincinnati, "Feature Tests of Scholar@UC," Last accessed: June 6, 2024. [Online]. Available: https://github.com/uclibs/ucrate/tree/develop/spec/features

[22] ——, "Scholar@UC user stories," Last accessed: June 6, 2024. [Online]. Available: https://github.com/uclibs/scholar_use_cases

[23] Y. Liu *et al.*, "RoBERTa: A robustly optimized BERT pretraining approach," *CoRR*, July 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1907.11692

[24] OpenAI, "GPT models," 2023, Last accessed: June 6, 2024. [Online]. Available: https://platform.openai.com/docs/models/gpt-3-5

[25] N. Niu *et al.*, "Advancing repeated research in requirements engineering: a theoretical replication of viewpoint merging," in *RE*, Beijing, China, September 2016, pp. 186–195.

[26] S. Sturmer *et al.*, "Eliciting environmental opposites for requirements-based testing," in *EnviRE*, Melbourne, Australia, August 2022, pp. 10–13.

[27] J. Guo *et al*, "Semantically enhanced software traceability using deep learning techniques," in *ICSE*, Buenos Aires, Argentina, May 2017, pp. 3–14.

[28] A. Fantechi *et al.*, "Inconsistency detection in natural language requirements using ChatGPT: A preliminary evaluation," in *RE*, Hannover, Germany, September 2023, pp. 335–340.

[29] A. D. Rodriguez *et al.*, "Prompts matter: Insights and strategies for prompt engineering in automated software traceability," in *SST*, Hannover, Germany, September 2023, pp. 455–464.

[30] C. Khatwani *et al.*, "Advancing viewpoint merging in requirements engineering: A theoretical replication and explanatory study," *Requirements Engineering*, vol. 22, no. 3, pp. 317–338, September 2017.

[31] M. R. Amin *et al.*, "Environmental variations of software features: A logical test cases' perspective," in *EnviRE*, Hannover, Germany, September 2023, pp. 192–198.