ISSN 2186-7437

NII Shonan Meeting Report

No. 207

Anti-patterns and Defects: Synergies, Challenges, and Opportunities

Fabio Palomba
Raula Gaikovina Kula
Shane McIntosh
Takashi Kobayashi

December 2–5, 2024

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Anti-patterns and Defects: Synergies, Challenges, and Opportunities

Organizers:
Fabio Palomba (University of Salerno, Italy)
Raula Gaikovina Kula (The University of Osaka, Japan)
Shane McIntosh (University of Waterloo, Canada)
Takashi Kobayashi (Institute of Science Tokyo, Japan)

December 2–5, 2024

Background and Introduction

Source code anti-patterns, also known as code smells, are poor design or implementation choices applied by programmers when evolving a software system that can eventually lead to the introduction of defects, i.e., incorrect software system behavior [1]. For decades, the software engineering research community has been actively investigating anti-patterns and defects in isolation. While the advances that the research community has made with respect to anti-patterns and defects are numerous (e.g., [2, 3]), there is room for each community to benefit from the other. Furthermore, in light of trending topics like generative Artificial Intelligence (AI) for Software Engineering, there is a growing need for these two sub-communities, which are naturally related to each other; however, few studies have investigated this intersecting boundary. Early results have investigated specific aspects, such as the defect-proneness of anti-patterns [1, 4], yet there is much more to be explored. As a consequence, several relevant key aspects are still unknown:

- Are we able to characterize the boundary between anti-patterns and defects?
- Can complex anti-pattern removal activities be prioritized based on the likelihood of cross-cutting from anti-pattern to concrete defects?
- How can anti-pattern detection inform defect identification and vice-versa? For instance, how advances in the use of machine learning approaches for defect prediction can be exploited to improve the prompt identification of anti-patterns?
- How do these challenges translate to the current trending topics in Software Engineering, such as generative AI?

The aim is not only to foster an exchange of ideas, but also to outline a clear set of concrete grand challenges and propose a roadmap for how those challenges can be met by future work.

Overview of the meeting

Our meeting brings together 27 experts from diverse regions around the world, each with unique experiences and qualifications from the Software Engineering domain. This diversity enables us to explore and understand software quality from a wide range of perspectives. The primary objective of this gathering is to establish clear definitions of software quality, reflecting the varied viewpoints represented by the participants. By doing so, we aim to dismantle the silos that have formed between different ideas of anti-patterns and defects, and how software quality is perceived in this current time. This collaborative effort will allow us to identify synergies, overcome challenges, and uncover opportunities, providing a clear vision for the direction of work in software quality research over the next 5-6 years.

Originally proposed in 2019, the idea of this Shonan meeting has changed immensely. Covid-19 and advancements of Large Language Models has led to a disruption in the field, especially with how developers now develop code. In this manner, the workshop had to address this elephant in the room, with the core question becoming, what is the current state, and how do anti-patterns and defects look like in the future. Based on these changes, organizers decided to propose a much large framework for the participants to work:

- User Perspective - From the user's perspective, software quality is assessed based on its ability to meet their needs, expectations, and preferences.
- Developers Perspective - From the developer's perspective, software quality focuses on maintainability, scalability, and code quality.
- Quality Assurance Perspective - The quality assurance (QA) perspective is centered on ensuring that the software meets predefined quality standards and functions as expected in all scenarios.
- Operator Perspective - From the operator's perspective, software quality is defined by its reliability, deployability, and ease of monitoring in a production environment.

Unlike previous Shonan meetings, the format of this event was designed to focus entirely on breakout sessions, fostering deeper engagement and collaboration. To further encourage relationship-building and idea exchange, all social events were scheduled during the initial days of the meeting, creating a foundation of camaraderie and trust before diving into focused discussions. The event then progressed with three intensive breakout sessions, culminating in concrete discussions aimed at actionable research agendas. These discussions were designed to outline collaborative ideas and projects that research teams could pursue together, ensuring that the outcomes of the meeting would have a tangible impact.

A notable feature of our breakout sessions was the inclusion of a walking session, which provided participants with an informal setting for closer, more personal discussions. This unique approach fostered a relaxed yet productive environment, allowing participants to engage in meaningful conversations that complemented the structured breakout groups. This blend of formal and informal discussions proved invaluable in building connections and enriching the depth of the meeting's outcomes.

Meeting Schedule

Check-in Day: December 1st 2024 (Sun)

- Welcome Reception

Day1: December 2nd 2024 (Mon)

- Introduction to Shonan and Participant Introductions.
- Discussion Topics Brainstorming
- Shonan Excursion to Enoshima and Banquet Dinner

Day2: December 3rd 2024 (Tue)

- Breakout Session 1
- Report on Breakout Session 1
- Breakout Session 2
- Report on Breakout Session 2

Day3: December 4th 2024 (Wed)

- Breakout Session 3 (Walking)
- Report on Breakout Session 3 (Walking)
- Concrete Ideas Session 4 (Breakout into groups on concrete ideas)
- Report on Concrete Ideas Session 4 (Summary and updates of the studies to be conducted)

Day4: December 5th 2024 (Thu)

- Wrapping Up, concrete ideas and follow-up on potential collaborations.

List of Participants

- Fabio Palomba, University of Salerno, Italy
- Raula Gaikovina Kula, The University of Osaka, Japan
- Shane McIntosh, University of Waterloo, Canada
- Takashi Kobayashi, Institute of Science Tokyo, Japan
- Brittany Reid, Nara Institute of Science and Technology, Japan
- Gemma Catolino, University of Salerno, Italy
- Thomas Zimmermann, University of California, Irvine, USA
- Kelly Blincoe, University of Auckland, New Zealand
- Valeria Pontillo, Vrije Universiteit Brussel, Belgium
- Earl Barr, University College London, United Kingdom
- Coen De Roover, Vrije Universiteit Brussel, Belgium
- Daniel German, University of Victoria, Canada
- Mahmoud Alfadel, University of Calgary, Canada
- Michele Lanza, USI Lugano, Switzerland
- Yutaro Kashiwa, Nara Institute of Science and Technology, Japan
- Hideaki Hata, Shinshu University, Japan
- João F. Ferreira, INESC-ID / University of Lisbon, Portugal
- Dario Di Nucci, University of Salerno, Italy
- Shinpei Hayashi, Institute of Science Tokyo, Japan
- Andrian Marcus, George Mason University, USA
- Takashi Ishio, Future University Hakodate, Japan
- Akond Rahman, Auburn University, USA
- Csaba Nagy, USI Lugano, Switzerland
- Moritz Beller, Meta, USA
- Chaiyong Ragkhitwetsagul, Mahidol University, Thailand

Overview of Breakouts

Breakout 1 - Antipatterns and Defects in Configurations

Participants: Akond Rahman, Moritz Beller, Yutaro Kashiwa, Andrian Marcus, Shane McIntosh, Dario Di Nucci, João F. Ferreira, Mahmoud Alfadel

Context : Riskiness of code changes that impact downstream revenue through outages. Configuration changes are much higher than that of regular code changes, much of which are LLM-generated Gradual rollout of configuration changes To determine the riskiness you rely on metrics, that are not in the same granularity as the production deployment Similar to code changes, configuration changes can be tangled within a single diff Metrics are hard to improve because of the scale of the development process at Meta that involves impacting the workflow of thousand developers Configuration changes are of two types: Python-based (e.g., changes in variable values) and Prioritizing configuration value changes is challenging Configuration testing: gradual rollouts, simulation, and canary deployment (e.g., replay production traffic and see breaks or not)

State of the art: Measure metrics to determine the oracle If a configuration change does not impact performance, perhaps this is a less risky set LLMs can be used for configuration validation (https://tianyin.github.io/pub/ciri.pdf) Existing solutions are limited based on scale Existing resources: NIST-glossary and NIST-document Semantics of configuration code changes sounds promising as per SOSP paper

Open challenges: What paths are promising to test configurations at scale? What paths should we avoid? What does it mean to test the entire configuration set for systems of systems? (long-term) What is the likelihood of this particular configuration change breaking ? Riskiness is a probability-based model and how do you convince the stakeholders that the risk-based solution will work? Especially in the cases of false positives and false negatives Configurations that span across microservices ... how do we make sure we are testing them well, e.g., with integration testing? What are the recurrent root causes for large-scale outages [long-term goal]

Proposed solutions: Categorize the configurations: declarative vs imperative, infrastructure vs product, Prioritize misconfigurations based on certain attributes, such as scheduling, routing, queueing, and region availability Automate testing activities as much as we can possibly with AI agents or LLMs Automated testing works to some extent for example for unit tests but what about simulating users? Simulation testing is considered a hack in the context of configuration testing that is distributed in nature. Can we understand the root causes of why simulation testing fails for testing configurations? Can AI assistants, i.e., assistants beyond LLMs use AI to generate valid configuration values?

Breakout 1 - Human-In-The-Loop

Participants: Chaiyong Ragkhitwetsagul, Fabio Palomba, Gemma Catolino, Kelly Blincoe, Michele Lanza, Thomas Zimmermann, Hideaki Hata, Takashi Kobayashi, Shinpei Hayashi

Context: The rise of Large Language Models (LLMs) is reshaping the traditional software engineering process, raising questions about the evolving role of humans, particularly developers. Traditionally, developers have been central to all stages of the software lifecycle, from requirements engineering (RE) to operation, maintenance, and evolution. However, LLMs are increasingly automating these stages, introducing new dynamics in software creation, while still posing risks such as technical debt, defects, and anti-patterns.

State of the art: Currently, humans remain crucial in requirements engineering, where their expertise drives the effective use of LLMs. The design phase is another area where developers maintain significant influence, shaping the project direction. As the project progresses, LLMs become more autonomous, executing tasks with decreasing human intervention. However, challenges persist in leveraging LLMs effectively, particularly in ensuring code quality, comprehensibility, and maintainability. Current tools lack sufficient mechanisms for evaluating and refining outputs from LLMs.

Open challenges: Several critical challenges arise in integrating LLMs into the development workflow. The team identified two main endpoints where developers would still be fundamental. First, *pull request and issue request assessment* face significant challenges. There is an absence of robust metrics to evaluate the quality of code generated by LLMs, which hampers developers' ability to make informed decisions. Additionally, the lack of effective visualization tools complicates the process of understanding and reviewing LLM-generated content. Second, *testing mechanisms* also present notable limitations. Current approaches for non-functional testing are inadequate, leaving gaps in assessing crucial qualities, e.g., performance and security, in the code automatically generated by AI assistants. In this respect, there remains a pressing need for human oversight to ensure that these aspects are thoroughly evaluated and aligned with the desired system requirements. Human involvement is critical not only for verifying the correctness and relevance of the test cases generated by LLMs but also for providing contextual judgment and addressing edge cases that automated tools may overlook. Such oversight can enhance the reliability and robustness of the overall testing process.

Proposed solutions: To address these challenges, a few potential solutions are proposed, focusing on enhancing quality assessment both in the prompts (inputs to LLMs) and in the generated contents by LLMs (outputs from LLMs) to better integrate human oversight and LLM capabilities. This can be done starting by understanding the metrics that can be used to describe the properties of the prompts (e.g., NLP-based metrics) and the metrics that can be used to evaluate the quality of the generated contents (e.g., code) by LLMs.

Breakouts 1 - Program Analysis: Present and in the Future

Participants: Coen De Roover, Valeria Pontillo, Csaba Nagy, Brittany Reid, Raula Gaikovina Kula, Earl Barr, Daniel German, Takashi Ishio

Problems As developers start to use LLMs to generate code, a major challenge is analyzing suggestions generated by large language models (LLMs). These analyses must contend with the assumption of whole-program analysis while dealing with the reality of partial, syntactically incorrect code. Potential solutions include employing compositional program analysis or adapting analyses designed for Stack Overflow snippets, although these approaches may require addressing the unique characteristics of placeholders used in code suggestions. Another short-term issue is the quality of suggestions provided for "low-resource ecosystems," which are often plagued by hallucinated or nonsensical code. Addressing this involves quantifying the quality of suggestions using ecosystem-specific confidence metrics and feeding these metrics back into program analyses. There is also the question of whether program analyses can influence prompt design, such as enabling prompts like "lower cyclomatic complexity, please," which may represent a significant technical challenge.

In the long term, the integration of LLM-generated suggestions into actual codebases requires a longitudinal analysis, similar to those conducted in Mining Software Repositories (MSR) studies. Key challenges include addressing non-determinism and ensuring traceability of the context, prompts, and the developer's own edits. Potential solutions include developing tooling to recognize generated code and maintaining its history for better accountability and analysis.

Opportunities In the short term, there is an opportunity to explore data-driven alternatives to traditional, engineering-heavy program analyses. This is particularly relevant in the diverse and ever-changing landscape of software development, where algorithmic support for every single feature or language ecosystem is impractical. Techniques highlighted in studies like "A Few Billion Lines of Code Later" could inform such approaches. Additionally, AI-enhanced program analysis could leverage natural language semantics to identify likely sources and sinks of sensitive information, enabling more sophisticated and flexible tools.

In the long term, the information conveyed by well-defined interfaces for black-box LLM-generated components presents a promising avenue for innovation. Enhancing program comprehension support for LLM users is also crucial, as they must now balance writing and reviewing code simultaneously. Finally, verifying contracts between LLMs and developers, and safeguarding the integrity of the remaining software, could create a more robust and trustworthy ecosystem for integrating LLM-generated code.

Breakout 2 - New Artifacts, Old Bugs

Participants: Brittany Reid, Dario Di Nucci, Raula Gaikovina Kula, Daniel German, Andrian Marcus, Gemma Catolino, Michele Lanza

Context: Developers are increasingly incorporating LLM usage into their software development workflows. As they write tests, deploy code etc. this necessitates the creation of new artifacts. Similar to the concept of Infrastructure as Code (IaC), where specifications are maintained as files under source control just like code, it's possible that data like LLM input, output, context or examples used in few-shot training could also be considered an artifact. Under this paradigm, we posit that new and similar issues may emerge in the handling of such artifacts. We ask: what kinds of new issues may emerge, and how could current state-of-the-art still apply to new types of artifacts?

State of the art: In current software development practice of Infrastructure as Code (IaC), everything that is created and put into operation during software development (library dependencies, source code, test cases, production metrics, software bill of materials (SBOMs) etc.) should be considered an artifact treated similarly to code. Additionally, there are already examples of leveraging code developed by others that must be trusted, for example with library usage or reuse of code from Stack Overflow. However, developers are currently generating parts of their programs using LLMs, but not recording which parts, or what input was provided to generate that part.

Open Challenges: Key challenges include:

- As new artifacts arise from the usage of LLMs, should versioned data like prompts, and few-shot learning examples become artifacts that are persisted?
- To what extent will these new artifacts impact program comprehension and team collaboration?
- "The context is the king." A kind of issue could be related to the difference between the expected code and the generated one. There is a new trade-off between the context size and costs.
- Nondeterminism of LLMs is an issue that hinders the reproducibility of the results.
- Will students go from "zero to hero" if learning basic concepts is becoming less and less relevant?
- Will we have a major software crisis due to developers strongly relying on LLMs, which will eventually not have new code to learn from?
- What are the effects of LLMs on developer-to-developer communication?
- Some developers do not like Copilot because it changes their job from designing code to reviewing code; what impacts will LLMs have on how developers see their jobs?
- What are the effects of LLMs as personal tutors on education?

Proposed solutions:

- In a world where code is fully developed by machines, software quality, e.g., comprehensibility, could radically change. Think about bytecode quality. However, code could also be developed hybrid, e.g., using LLMs to code methods, write test cases, and document code portions.
- We need to educate students on the pros and cons of using LLMs to deliver SE tasks.

Breakout 2 - Dynamic Analysis, GenAI and vice versa

Participants: Takashi Ishio, Takashi Kobayashi, Yutaro Kashiwa, Hideaki Hata, Akond Rahman, Mahmoud Alfadel, Csaba Nagy

Context: How can we make use of LLMs for dynamic analysis problems such as debugging and log analysis? (AI for dynamic analysis) How can we make use of dynamic analysis for analyzing generated code? (Dynamic analysis for AI) Note that dynamic analysis for the development of generative AI is out of scope.

State of the art: Established dynamic analysis problems are now being replicated with generative AI. We learned about prior work that has used generative AI that involves dynamic execution of software artifacts as follows.

- Debugging: LLM is employed to automate fault diagnosis tasks. The accuracy of fault diagnosis by ChatGPT can be improved by using program execution traces in prompts [5].
- Crash report analysis: LLM is employed to analyze the root cause of a crash in code and environment and automatically fix the problem [6–8].
- Obfuscation detection: LLM is employed to detect the location of the TADA (anti-dynamic analysis) implementation in the code, to help reverse engineers place breakpoints used in debugging [9].
- Log parsing and anomaly detection: LLM is a promising tool to automatically parse log messages [10]. It is also applied to log anomaly detection [11].
- Fuzzing: LLM is also employed as a Fuzzer to analyze the behavior of a program [12, 13].
- Assessment of generated code: A study compared the performance (speed and memory consumption) of generated code with those of human-written code and reported that some of generated codes had performance problems [14].

Open Challenges: Comprehension is more important for generated source code. Dynamic information can help in supporting related tasks. The discussion identified open challenges as follows:

- How can we analyze generated code? It might be incomplete.
- How can we understand the behavior of generated code? The generated code itself might be difficult to understand.
- How can we deal with large execution traces for LLMs? How can we summarize execution traces so that LLMs and developers can understand?

Breakouts 2 - Developer's Interactions and Anti-Patterns:

Participants: Moritz Beller, Kelly Blincoe, Shinpei Hayashi, Chaiyong Ragkhitwetsagul, Fabio Palomba, Valeria Pontillo, Thomas Zimmermann

Context: After discussing the proposed solutions of the session ***Human-In-The-Loop***, we observed that in some phases of software development, i.e., Requirements Engineering and Design Phases, developers are required to maintain a higher level of focus compared to other phases. This raises questions about the ideal level of involvement throughout the process: should it remain consistent, or should it adapt based on the phase? Developers must also understand how their engagement levels should change in response to varying demands of the development lifecycle. Additionally, the interaction between developers and Generative AI could introduce new socio-technical challenges, such as redefining anti-patterns and assessing their implications on trust and collaboration.

State of the art: Traditional anti-patterns, often aligned with code smells, primarily target functional aspects of the generated code. However, in the context of Gen AI, anti-patterns extend beyond traditional definitions to include non-functional requirements, such as usability, security, and performance. Current research has yet to explore specific anti-patterns related to Large Language Models (LLMs), particularly those emerging from the interaction between developers and AI systems. Existing tools focus on code smell detection, bug identification, and defect prediction but do not adequately address anti-patterns specific to LLM-generated artifacts or their socio-technical implications.

Open challenges: Key challenges include:

- Defining anti-patterns specific to LLMs, not limited to code but encompassing non-functional requirements.
- Establishing trust metrics based on these anti-patterns.
- Detecting and addressing anti-patterns in the interaction between developers and Gen AI systems.
- Managing the inherent non-determinism of LLM outputs, which complicates the consistent identification of issues.

- Developing metrics to assess prompt quality, such as length, entropy, grammar, structure, and atoms of confusion.
- Integrating these metrics into a framework capable of analyzing both prompts and responses for anti-patterns.

Proposed solutions: We propose a two-layered framework for analyzing and mitigating anti-patterns:

1. *Developer-to-LLM Layer:* This layer focuses on prompt verification to identify anti-patterns within the input. Basic metrics, including NLP metrics (length, entropy, grammatical accuracy, and structural coherence), as well as atoms of confusion, will be computed to assess prompt quality.
2. *LLM-to-Developer Layer:* This layer evaluates the LLM's responses for anti-patterns. Techniques such as code smell detection, bug prediction, defect analysis, and security assessments will be employed to measure the quality and reliability of the generated output.

Breakouts 3 (Concrete Ideas)- Emergence of Policy as Code in the Era of Infrastructure Automation

Participants: Mahmoud Alfadel, Akond Rahman, Coen De Roover, Yutaro Kashiwa, Raula Gaikovina Kula

The discussion centered on the potential of *Policies as Code (PaC)* to enhance the management and automation of software infrastructure configuration. PaC encodes governance rules (e.g., compliance, resource limits, and access control) as executable code, allowing automated enforcement and validation. This research initiative aims to explore the capabilities, adoption, and challenges of PaC within infrastructure configuration.

Research Focus

- **Understanding PaC Adoption:** How widely are PaC tools (e.g., Open Policy Agent, HashiCorp Sentinel) used in real-world projects? What types of policies (security, compliance, optimization) are most common?
- **Analyzing Policy-Configuration Interactions:** How do configuration changes impact defined policies? Can patterns in policy violations or updates be systematically mined?
- **Challenges in PaC Development:** What are the technical and organizational barriers to adopting PaC? How can policy drift and ambiguities in translating intent to code be addressed?

Proposed Research Methodology

- **Empirical Study of PaC Projects:** Survey open-source projects that use PaC tools to understand adoption patterns, policy types, and validation practices.

- **Data Mining for Policy Insights:** Analyze repositories for change patterns in policies and their associated configurations. Identify trends in policy violations, updates, and their impact on system stability.
- **Prototyping and Tool Enhancement:** Develop prototypes to automate policy generation and validation, integrating with tools like Kubernetes or Terraform. Explore LLMs for policy recommendation and conflict detection.

Challenges and Opportunities

Challenges: Policy drift, ambiguity in policy definition, and scalability in dynamic systems.
Opportunities: Automating policy creation, aligning configurations with governance standards, and integrating PaC into DevOps pipelines.

Next Steps

- Conduct a survey of open-source projects using PaC tools to gather insights on practices and challenges.
- Develop benchmarks to evaluate PaC tools in terms of effectiveness and coverage.
- Initiate a case study to observe policy evolution and its impact in real-world infrastructure projects.

Expected Outcomes: This research will establish a foundational understanding of PaC for infrastructure configuration to enhance its adoption and reliability in modern software systems.

Breakouts 3 (Concrete Ideas) - The Role of AI in the Software Development Cycle

Participants: Daniel German, Moritz Beller, Shane McIntosh

Context: When discussing AI and its impact on software development, conversations often suffer from misunderstandings that impede progress. Terminology is highly generic, which inherently creates confusion.

State of the art: While some research has explored the past state and future roadmaps of AI-supported Software Engineering (SWE), these studies have focused on isolated examples and far-future projections. We argue that significant progress has been made in the field in the last two years, having begun to transform SWE profoundly. At the same time, despite this momentum, we contend that SWE will not be entirely dominated by AI in the near or mid-term future.

- https://arxiv.org/pdf/2410.20791
- https://arxiv.org/pdf/2410.06107

- https://dl.acm.org/doi/abs/10.1145/3663529.3663849
- https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9360852

Open challenges: In this discussion group, we set out to tackle this challenge by (a) providing a framework to structure thinking and (b) creating a common nomenclature.

Proposed solutions: We propose to align discussions along the following dimensions:

- Stakeholders
  - Traditional software stakeholders plus...
  - Model creators
  - Model operators
  - AI itself
- Agency
  - Who initiates?
    - AI itself
    - Developer
    - User
  - Who approves AI decisions?
    - AI itself
    - Developer
    - User
  - Stage of software cycle
  - Interaction with the AI: The actions of an AI can be classified in two major dimensions:
    - Interaction trigger: what initiates the actions of the AI
    - Agent initiated (the AI is continuously running and deciding when to act)
    - Developer initiated
    - End-user initiated: this can only occur at run-time when the End-user is in control
    - Interaction type:
      - No-human-in-the-loop: once started, the AI does not need any
      - interaction with the human (request, or approval)
      - Explicit-request. A human makes a explicit request via either an API or a prompt
      - Implicit-request: The AI is observing the actions of the human and suggests actions that need to be approved by the human.

    - Size of the intervention
        - Program
        - ...
        - Component
        - Function
        - Statement

AI as a component: Software will continue to be component based. AI will be instantiated in software systems (during development and during run-time) via components with clear interfaces. In a way, this is a natural evolution from Agent components.

AI components will greatly increase the challenges present in Component and Agent-based software engineering. For example:

- A component requires an interface: input/output
    - Prompts vs traditional APIs
    - Text output vs structured output
- AI components are non-deterministic in nature
    - They are expected to make errors
        - how do we assess their quality?
        - Short term and long term stability (do they behave the same over time)?
        - Versioning
- Components will be used by different stakeholders:
    - Software developer
    - End-user, and this affects the state of the software development
- Who operates the models:
    - Where does the model run?
        - As part of the end-user application?
        - As a service (running remotely and connected to the end-user application using the Internet)
    - When the model is operated by a different entity,
        - What is the service contract? Quality? Cost? Intellectual property issues?
    - Retraining issues

Breakouts 3 (Concrete Ideas) - In Search for Metrics for Prompt Quality Assessment

Participants: Gemma Catolino, Shinpei Hayashi, Csaba Nagy, Chaiyong Ragkhitwetsagul, Brittany Reid, Fabio Palomba, Valeria Pontillo, Thomas Zimmermann

Context: The discussion highlighted the need to create and propose tailored metrics for evaluating developers' interactions with LLMs. This involves LLMs outputs at different phases of project development, recognizing that challenges such as antipatterns may impact entire conversations or isolated parts of them. This research opens an avenue for further exploration, particularly in understanding the granular effects of such antipatterns on LLM-generated content.

State of the Art: There is limited research on tailored metrics to evaluate developer interactions with LLMs, particularly across different phases of project development. The DevGPT dataset [15], which links prompts to responses, offers a foundational resource for such evaluations. Natural language processing (NLP) techniques, such as those used in readability studies, unstructured text mining, and quality assessments in requirements engineering, provide inspiration for defining prompt and response metrics. In software engineering, established metrics like code smells and technical debt indicators are commonly used to assess code quality but are not yet systematically applied to LLM-generated outputs.

Open Challenges: Key issues identified in the outputs generated by LLMs include non-determinism, completeness, correctness, context limitations (such as forgetting information or missing links in multi-interaction scenarios), and the overall amount of detail and quality of the responses. These problems highlight the importance of evaluating both prompts and responses systematically and comprehensively. The discussion revolved around this point.

Proposed Solutions: A proposed preliminary study would focus on individual prompt-level analysis to investigate whether specific textual patterns influence the quality of responses. Metrics to evaluate prompts could include size, the number of common words, the number of code elements, cosine similarity (to measure alignment with contextual prompts), readability, vocabulary richness, typos, abbreviations, and the presence of identifiable prompt patterns. Inspiration for these metrics could be drawn from natural language processing (NLP) techniques, such as those used in readability studies, unstructured text mining, and requirements engineering quality research. The DevGPT dataset [15], which includes connections between prompts and responses, offers a practical foundation for this analysis. For measuring responses, the discussion suggested metrics to assess completeness (e.g., examining comments in pull requests), correctness (e.g., comparing generated code to actual code), and the amount of detail (e.g., size, number of comments). Quality could be further evaluated through established metrics for software quality, such as code smells and technical debt indicators. This approach, leveraging the DevGPT dataset and a combination of NLP and software engineering metrics, aims to systematically assess the relationship between prompts and responses, providing insights into optimizing LLM use for project development tasks.

In particular, once the quality attributes of prompts and the response attributes have been measured, a statistical analysis is envisioned to explore their relationship. In this analysis, quality attributes of prompts will be treated as independent variables, while response attributes will serve as dependent variables. To address the non-deterministic nature of LLMs, prompts will be executed multiple times so differences in responses can be computed to understand variability and reliability. The statistical analysis will incorporate variations in quality attributes, achieved through prompt mutations. This iterative approach aims to capture the effects of prompt quality on the consistency, completeness, and correctness of responses over multiple iterations, offering a framework for evaluating and improving the use of LLMs in software development tasks.

A potential follow-up study is to investigate the effects of the changes that are made to the prompts, while still preserving the prompt's intention, on the quality of the responses. The results from the previous phase can be used as the baseline for comparison. Some of the text perturbation techniques introduced by Ribeiro et al. [16] can be adopted, such as negation, robustness (swapping characters to create typos), and vocabulary (i.e., replacing words by their synonyms). By submitting the perturbed versions of the prompts, we can identify modifications that positively impact the quality of the prompt responses. The findings from this study can lead to an automated prompt-improving technique or tool.

Breakouts 3 (Concrete Ideas) - LLMs for Specification Generation

Participants: Earl Barr, Takashi Kobayashi, Hideaki Hata, João F. Ferreira, Dario Di Nucci, Takashi Ishio

When available, specifications precisely document code and allow us to automatically prove properties of our code. Despite their value, developers rarely write them. We posit that this is due to the cost of writing them. LLMs are already revolutionising development; here, we ask whether the LLM revolution will extend to specification writing by reducing their cost.

We will start by identifying which specification languages LLMs generate candidate specifications that speed a developer's creation of a "good enough" specification. We apply the "good enough" principle because specifications will vary by task and context. The LLM's candidate specifications could even be incorrect or arbitrarily imprecise because our goal is to produce candidate specifications that will speed developer's formulation of the specification. In short, we view LLMs as bicycles for the mind, as tools to speed and augment, not replace, human reasoning.

On languages which appear frequently in pretraining data, LLMs repeatedly demonstrate shockingly effective performance on many tasks. On languages for which we lack data and resources, LLMs tend to fall short of this high bar. These languages are low-resource languages. Because developers tend not to write specifications, it is not clear whether specification languages are low-resource languages. We aim to answer the research question: Which specification languages are low-resource languages?

There are many axes along which to answer this question. We will focus our inquiry on three axes: specification languages, input type and granularity, and formality of the specification language. LLMs are omnivorous: they consume and produce arbitrary sequences. Of the many input types and granularities LLMs can consume, we will start with code and its natural granularities — i.e., file, method, and loop. Specification languages vary widely in formality, from coarse-grained weak-type systems to rich program logics. Along this axis, we look to establish a correlation between the features of specification languages and their data availability. Specifically, are low-resourced specification languages more expressive or more strict?

This is a programme. It will start empirically, as sketched above, to get the lay of the land. Promising directions include:

- All LLM work suffers from a largely unacknowledged, validity threat: that the test data has already appeared in the pretraining data, so that performance is due to memorisation, not interpolation. We will develop new techniques to generate valid code not present in pretraining data.
- User studies to understand to what extent the proposed solution facilitates developers in writing formal specifications at different levels of granularities.
- Empirical studies aiming to analyse the effectiveness of LLM-based specification generation for property-based testing and other applications.
- Case studies to verify to what extent the automated generation of the formal specification can be employed in the context of smart contract development in Solidity, eventually featuring industrial use cases.
- Building tooling that leverages our empirical findings. For example, if we identify code features that correlate with LLMs effectively generating specifications for that code, we could build a classifier that enables us to apply LLMs only to such code.
- Support methods based on the generated specifications, such as leveraging the generated specifications as input for automated test generation.

Next Steps: We will schedule regular meetings to progress this project. We will work to identify relevant funding sources for this research programme across Portugal, Italy, the United Kingdom, and Japan, e.g., JSPS International Joint Research Program, JST ASPIRE, EU MSCA Doctoral Networks, Horizon Europe.

Summary of discussions and new findings

One clear finding is that Large Language Models (LLMs) are here to stay (evident by the large portion of the plenary session discussing its usage), becoming integral to software development and other domains. While there is some push-back regarding their widespread adoption—driven by concerns over trust, ethics, and over-reliance—there remains significant research and practical work outside the realm of generative AI. For instance, issues like mis configuration and its management continue to demand attention, highlighting that not all advancements hinge on LLMs or generative AI disruption. These areas underscore the need for a balanced focus across traditional and emerging challenges in the field.

The discussions encompass a range of critical topics in software development. These include Antipatterns and Defects in Configurations, examining recurring configuration mistakes; Human-In-The-Loop for New Dynamics in Software Creation, addressing the balance of innovation with risks like technical debt and anti-patterns; and Program Analysis: Present and in the Future, focusing on evolving techniques to address both traditional and emerging challenges. Additional topics explore New Artifacts, Old Bugs, highlighting the persistence of longstanding issues in the context of generative AI; Dynamic Analysis, Generative AI and Vice Versa, delving into the mutual influence between these technologies; and Developer's Interactions and Anti-Patterns, investigating the role of developer behaviors in shaping software quality. Four potential concrete research ideas that arose from this meeting is as follows:

- Potential for studying Policy as Code.
- Investigate the role of AI in the Software Development Cycle.
- Software Quality Assurance applied to Prompting of generative AI technologies.
- Formal Specifications applied to LLMs and other generative AI technologies.

Participants drew inspiration from various concept designs and understanding how the developer would interact with an LLM to get suggestions. Thus anti-patterns and defects may also creep into the new artifacts now that include both the prompt and the outputted code.

Identified points of collaboration and future directions

- Large Language Models are here to stay! The Era of LLMs has arrived, reshaping the developer workspace whether experts like it or not. This is evident from the fact that three out of four proposed research topics are related to generative AI.
- Anti-patterns and defects will persist. Experts agree that traditional research topics, such as anti-pattern and defect detection for software quality assurance, remain crucial. While these issues may evolve in the future, the core problems researchers must address remain fundamentally the same.

Participants will continue to work on collaborations via the communication channel (discord). All participants agreed to add the Shonan 207 meeting as acknowledgment in future publications. The discussion points from this report may be extended for an article that could be uploaded as an arxiv technical document.

References

[1] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012.

[2] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Trans. Softw. Eng.*, 43(11):1063–1088, November 2017.

[3] S. Hangal and M.S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 291–301, 2002.

[4] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Softw. Engg.*, 23(3):1188–1221, June 2018.

[5] Takafumi Sakura, Ryo Soga, Hideyuki Kanuka, Kazumasa Shimari, and Takashi Ishio. Leveraging execution trace with chatgpt: A case study on automated fault diagnosis. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 397–402, 2023.

[6] Maroa Mumtarin, Md Samiullah Chowdhury, and Jonathan Wood. Large language models in analyzing crash narratives – a comparative study of chatgpt, bard and gpt-4, 2023.

[7] Priyanka Mudgal, Bijan Arbab, and Swaathi Sampath Kumar. Crasheventllm: Predicting system crashes with large language models, 2024.

[8] Xueying Du, Mingwei Liu, Juntao Li, Hanlin Wang, Xin Peng, and Yiling Lou. Resolving crash bugs via large language models: An empirical study, 2023.

[9] Haizhou Wang, Nanqing Luo, and Peng LIu. Unmasking the shadows: Pinpoint the implementations of anti-dynamic analysis techniques in malware using llm, 2024.

[10] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Peter Chen, and Shaowei Wang. LLMParser: An Exploratory Study on Using Large Language Models for Log Parsing . In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1209–1221, Los Alamitos, CA, USA, April 2024. IEEE Computer Society.

[11] Wei Guan, Jian Cao, Shiyou Qian, and Jianqi Gao. Logllm: Log-based anomaly detection using large language models, 2024.

[12] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[13] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 423–435, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Sila Lertbanjongngam, Bodin Chinthanet, Takashi Ishio, Raula Gaikovina Kula, Pattara Leelaprute, Bundit Manaskasemsak, Arnon Rungsawang, and Kenichi Matsumoto. An Empirical Evaluation of Competitive Programming AI: A Case Study of AlphaCode . In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 10–15, Los Alamitos, CA, USA, October 2022. IEEE Computer Society.

[15] Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. DevGPT: Studying developer-chatgpt conversations. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 227–230. IEEE, 2024.

[16] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. Beyond accuracy: Behavioral testing of NLP models with CheckList. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 4902–4912, Online, July 2020. Association for Computational Linguistics.