GEMS: Generative Expert Metric System through Iterative Prompt Priming

TI-CHUNG CHENG, University of Illinois Urbana-Champaign, USA
CARMEN BADEA, Microsoft Research, USA
CHRISTIAN BIRD, Microsoft Research, USA
THOMAS ZIMMERMANN, Microsoft Research, USA
ROBERT DELINE, Microsoft Research, USA
NICOLE FORSGREN, Microsoft Research, USA
DENAE FORD, Microsoft Research, USA

Across domains, metrics and measurements are fundamental to identifying challenges, informing decisions, and resolving conflicts. Despite the abundance of data available in this information age, not only can it be challenging for a single expert to work across multi-disciplinary data [22], but non-experts can also find it unintuitive to create effective measures or transform theories into context-specific metrics that are chosen appropriately [11]. This technical report addresses this challenge by examining software communities within large software corporations, where different measures are used as proxies to locate counterparts within the organization to transfer tacit knowledge. We propose a prompt-engineering framework inspired by neural activities, demonstrating that generative models can extract and summarize theories and perform basic reasoning, thereby transforming concepts into context-aware metrics to support software communities given software repository data. While this research zoomed in on software communities, we believe the framework's applicability extends across various fields, showcasing expert-theory-inspired metrics that aid in triaging complex challenges.

1 INTRODUCTION

Across the industry, many engineering teams express a desire to understand and improve their software practices [27]. However, consistently and accurately measuring software engineering has remained elusive [8]. Measuring engineering's processes, artifacts, collaborations, and even impact can be challenging. Yet attempts to create measures of software engineering have always been around: most computer science students have seen the picture of Margaret Hamilton next to a stack of paper that stands taller than her, containing the printed code she wrote for the Apollo program. While these stacks of paper do not represent the impact of her work, they are some symbols of her work (the authors maintain that lines of code, or reams of code printouts, are not good metrics for measuring coding output).

Many engineering teams and organizations struggle to understand and improve their software practices. When faced with questions about impact or collaboration, what data should be used? When faced with questions about software, many engineers resort to data that is easily available and instrumented in their systems, for example, pull requests and commits (e.g., [4]). While these data can be useful in many cases, they are not always suitable proxies. For example, the number of pull requests can miss the complexity or difficulty of the work done; in another example, the best solution may actually delete code, which would reflect "negative" productivity if simply counting lines of code. In our experience, we have seen practitioners make these mistakes quite often, reaching for available data rather than taking a thoughtful approach to operationalizing the concept they are trying to measure and test. This is understandable! They are trained

Authors' addresses: Ti-Chung Cheng, University of Illinois Urbana-Champaign, Urbana-Champaign, USA, tcheng10@illinois.edu; Carmen Badea, Microsoft Research, Redmond, USA, cabadea@microsoft.com; Christian Bird, Microsoft Research, Redmond, USA, cbird@microsoft.com; Thomas Zimmermann, Microsoft Research, Redmond, USA, tzimmer@microsoft.com, cbird@microsoft.com; Robert DeLine, Microsoft Research, Redmond, USA, rdeline@microsoft.com; Nicole Forsgren, Microsoft Research, Redmond, USA, niforsgr@microsoft.com; Denae Ford, Microsoft Research, Redmond, USA, denae@microsoft.com.

as programmers and engineers, not as researchers. But the impacts of these mistakes can be significant for organizations and developers: For example, if work effort is measured incorrectly (whether coding, code reviews, debugging, or other invisible work such as unblocking others' work), it can lead to poor resource allocation, bad work design, burnout, and attrition.

In the face of limited experience designing and conducting analyses and, furthermore, limited data availability, many engineering teams still want to improve. One common way to do this is by learning from others [20]. However, this can still pose challenges because knowledge transfer is often most useful when done between similar contexts [20]; in engineering, this could be work processes, technology stacks, or programming languages used.

To accomplish this, individuals and teams need to locate the right person to pose the right questions to, often beginning by relying on local networks as early computer-supported cooperative work research by McDonald and Ackerman [23] illuminated the complex processes of expertise identification and selection when locating experts to accomplish a task. They underscored the necessity for individuals to identify the skills and expertise of others before consulting the most suitable expert. In software developer communities this challenge is further compounded by the scale and complexity of software systems and large-scale collaboration. Software engineers and technical leads often have ambiguous queries or objectives that require specific skill sets and search heuristics to locate key information to achieve these goals. For instance, questions like: "How should I improve my team's new hire experiences?" necessitate a nuanced understanding of developer operations, organizational management, processes, and communication, thereby challenging the refinement of questions and the narrowing of search criteria, especially when incorporating non-technical search parameters [16].

Subsequent studies, like those by Yarosh et al. [33], investigated how to assist individuals in locating these experts. However, many challenges faced by individuals cannot be addressed by a single expert. Instead, solutions often require the collective expertise and guidance of small communities or project-based teams. For example, a security developer might want to learn, "How can I design test cases that are ethical and just?" which would benefit from engaging with a panel of colleagues experienced in both ethics and security protocols.

Moreover, finding counterparts inside a company or developer community that understand the context of a given question is fraught with difficulty. Prior research, such as that by Elliott and Scacchi [7], highlights the emergence of subcultures and common jargon among developers, leading to the formation of new sub-communities as organizations expand. This phenomenon poses a challenge for semantic matching, as the phrases and contexts embedded in questions require a nuanced and deeper understanding. In addition, as organizations and developer communities grow and change, an individual's personal network is unlikely to keep up with the new growth, creating inherent limitations to "simply" asking someone for help. That is, even if they know the right questions to ask, they may not know the people exist, or the people they would have asked may have left the organization or community [19].

With these challenges in mind, we explored the following research question: **How can we leverage Large Foundation Models (LFMs) to match teams within software corporations to improve developer team performance?**

To answer this question, we explored and developed a prototype targeting software communities to automate expertise identification and selection within these software communities. We built this prototype using open-source software communities in order to support replication and eliminate privacy concerns. The results from our prototype identify metrics used for expert identification, thus providing greater visibility into the process, and are likely to transfer across domains (e.g. teams within software corporations), making it highly generalizable.

This technical report begins in Section 2 by defining the problem statement. Section 3 elaborates on the technical design and architecture of this generative-based system. We present initial outcomes and case examples in Section 5 and discuss the strengths, weaknesses, and opportunities of developing this prototype in Section 6.

This technical report discusses two major contributions: (1) this work demonstrates it is possible to leverage LFMs to create generalized and scalable systems to identify expertise with a particular goal in mind (in our explorations, improving developer productivity), and (2) we highlight an architectural approach and several key techniques that enable an LFM-powered system to provide usable results grounded in expert literature. We believe that incorporating LFMs is an innovative approach to addressing the challenges of expertise identification and community matching, and shows promise in helping individuals and teams in OSS communities and corporate networks identify relevant expertise across weak ties.

2 PROBLEM DEFINITION

We begin by splitting the research question: **"How can we leverage Large Foundation Models (LFMs) to match teams within software corporations to improve developer team performance?"** into three key components: (a) enhancing developer team performance, (b) team matching, and (c) the application of LFMs.

To address the improvement of developer team performance, we referenced prior literature covering decades of research which highlights challenges such as individual information needs during software engineering tasks [18], communication issues within and between software teams, and the complexities of navigating software systems [24]. Furthermore, efforts to define holistic measures for assessing developer productivity are noted [8]. Despite the valuable directions provided by these studies, the development of tailored solutions for specific team challenges remains prohibitively expensive. We define a high-level software engineering challenge as C, which may be so ambiguous due to the broad nature of the challenge that an individual would not be able to come up with solutions without in-depth research into its specifics. We introduce a goal metric (M_g) as a proxy for optimizing the resolution or mitigation of C. For the purpose of our system prototype, we limit C to a single M_g, while acknowledging that addressing C fully may require improving multiple goal metrics.

Understanding that numerous factors might influence M_g, we identify a set of supporting metrics (M_s) believed to impact M_g indirectly or directly. It's crucial to recognize M_g and M_s not as definitive metrics, but as proxies for better measurement and quantification of the targeted metrics.

In our efforts to improve developer team productivity, we adopt the communities of practice model advocated by Lave [20], which fosters regular interaction among individuals with a common interest or concern. This approach proves beneficial, particularly when addressing challenges that stem from institutional or tacit knowledge, which are not easily conveyed through documentation, or widely understood by general experts. We propose not just a match but a partnership between two teams: T_x, requiring support to address C, and T_y, a team that is deemed capable of providing support and improvement of M_g for team T_x.

Furthermore, we aim to harness LFMs for synthesizing and utilizing a broad spectrum of text-based information to develop metrics covering M_g and the set of M_s that helps identify T_x and T_y. This text-based information includes but is not limited to: code artifacts, system documentation, markdown, git commits, and measures of interactivity. With the advancements in LFMs [6], we leverage their contextual understanding, built-in logical reasoning capabilities, and ability to develop code. The LFM-based prototype system we developed is denoted as Generative Expert Metric System (GEMS).

With these definitions established, given available data (D), we articulate that the prototype demonstrates the system's ability to facilitate the following pipeline: $GEMS(D, C) \rightarrow (M_g, T_x) \rightarrow ((M_{s1}, M_{s2}, \ldots, M_{sn}), T_y)$.

3 APPROACH AND SYSTEM DESIGN

In this technical report, we developed a Generative Expert Metric System (GEMS) that accomplishes the user-specified task or goal specified in Section 2. It can successfully match two teams T_x and T_y that the system identifies, where T_x is identified as the team that most needs improvement with respect to the goal, and T_y is the best fit for supporting T_x toward achieving the goal. We detail how GEMS works, starting with an overview and followed by implementation details.

3.1 System Overview

To construct this system, we implemented multiple LFM-powered agents, orchestrated using an existing orchestration framework called AutoGen [30]. Additionally, we used Guidance [2], the OpenAI API, and a MySQL database in the implementation. The database stores the data used to test our prototype, representing common information available to developer teams. We used software artifacts from open-source software (OSS) repositories including source code, commit details, and discussion threads. Each agent is designed and prompt-engineered to complete specific operations; see system overview in Figure 1 below. GEMS consists of the following components:

Lead Orchestrator Acts as the primary interface and memory store, coordinating between the user, other agents, and the database to produce a final recommendation.

Programmer Crafts database queries to retrieve relevant information. Given a task, it designs proxies or refines a search based on its knowledge of the database.

Expert Leverages domain-specific knowledge of a human expert's research outcomes to make logical suggestions for GEMS. A breakdown of the expert agent is available in Figure 2

Judge Employs well-established decision-making algorithms to finalize the team or community match, ensuring the recommendation aligns with the user intent specified.

The scenario we used to drive the GEMS implementation is one where the user specifies the high-level goal G, e.g. improve developer on-boarding process, and expects team T_x to be identified, followed by team T_y that is the best-suited team or community match to drive T_x's improvement with regard to goal G. Once the user submits the high-level goal to the GEMS, the Lead Orchestrator proceeds with the following two tasks:

(1) First, the Lead Orchestrator attempts to define a main goal metric (naively, i.e., without external information) that will serve as a proxy for the specified high-level goal and will be used to identify the target team (T_x) that is the worst performer with regard to the specified goal. The Lead Orchestrator presents this metric as a function name and a description of what it should measure.
    This information is then passed to a Programmer agent that will either select from a predefined set of functions available in GEMS that matches closest to the Lead Orchestrator's goal or construct a new function to implement this metric. The Programmer agent will return the resulting function to the Lead Orchestrator and the subsequently computed metric values for all teams within the database. Intuitively, this process is similar to a manager in a corporation asking each of their direct reports to provide a specific metric score for each team under their management to make an initial evaluation.

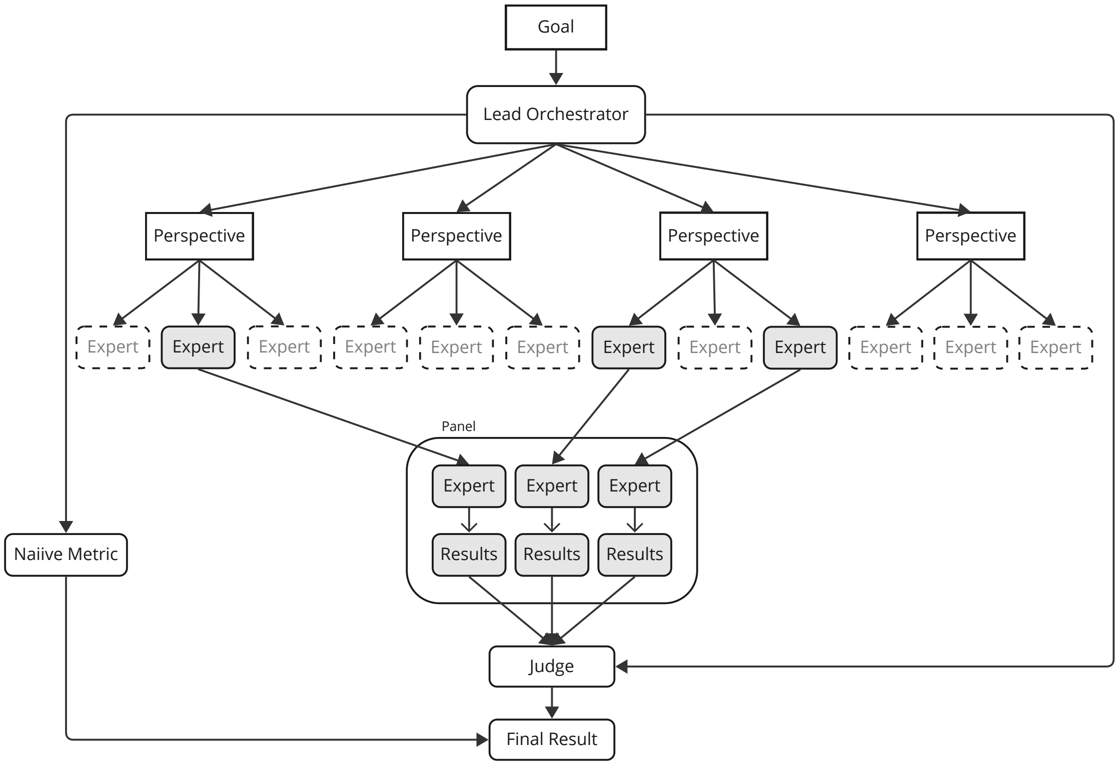

Fig. 1. System Overview. The system begins with the Lead Orchestrator defining a naive metric to identify Team T_x. For simplicity, we omit the details of how this metric was generated. The Lead Orchestrator subsequently forms a panel of experts augmented through perspectives relevant to the given goal. The experts generate teams' results that pass through to a judge agent to form a decision. A final result is then aggregated.

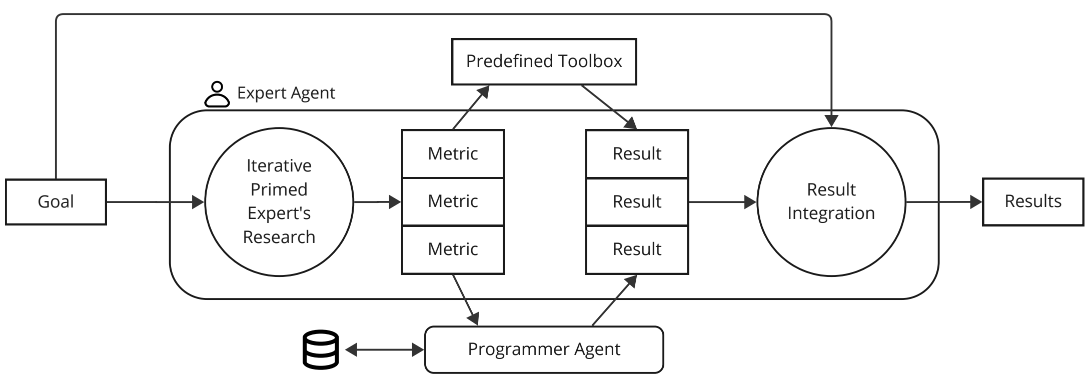

Fig. 2. Expert Agents. The agents are 'primed' by the given expert's research contributions to the field. This model is inspired by the generated knowledge prompting technique proposed by Liu et al. [21].

(2) Second, the Lead Orchestrator identifies relevant disciplines that provide insights toward the user-specified goal. Based on these disciplines, it generates and then judiciously assembles a panel of real-life experts in these disciplines, considering the panel expert selection criteria specified by the user, if any. The list of identified experts is shared with the user. Intuitively, this is a decision-maker assembling a panel of experts in relevant disciplines to tackle a complex problem.

The GEMS system then asks each expert agent (a proxy for a real-life expert) to independently suggest several factors that could positively influence the main goal metric based on each identified domain expert's knowledge. The factors identified by the expert agent as relevant to the matching decision are presented to the user.

These factors are then passed to a Programmer agent as a function name and function description for each factor. The Programmer agent will either generate an implementation for this function on the fly, select from a list of predefined functions, or execute it for all teams in the database (minus team T_x). The expert agent will then collect the computed values and use them to generate a list of team rankings indicating the best team match(es) they think can positively influence the target team toward improving the target metric/goal. A short explanation generated by each expert agent, followed by the team rankings, will be passed back to the Lead Orchestrator. Intuitively, this phase identifies traits that can affect the main metric from different lenses and generates a list of potential team matches.

Finally, the GEMS system passes all the above information to the Judge agent. This agent takes into account any user-specified preference aggregation mechanism when aggregating the ranking information coming from the expert agents. Examples of such mechanisms include approval voting, 1 person 1 vote, or rank choice voting. The judging agent makes a decision and final selection, and the final team match is passed back to the Lead Orchestrator. This completes selecting the counterpart team T_y denoted in the pipeline. The final recommended team matching result is then presented to the user.

3.2 Implementation Details

In this subsection, we break down intuitions and provide implementation details for used across multiple expert agents and judges.

3.2.1 Expert Agent and Iterative Prompt priming. Expert agents in GEMS have two tasks to perform:

(1) Design and define several metrics to measure and reflect on the given goal;
(2) Comprehend the results of these metrics to generate an interpretation of the aggregation of these results.

To achieve Task 1, we developed and implemented a prompt engineering technique that we called ***iterative prompt priming*** for expert agents. This process is inspired by [21], where knowledge is provided before answering a question to elicit more accurate responses. In the original method, the model is prompted to generate related pieces of knowledge before using this generated knowledge to answer the given question. This approach could not complete our goal, given the dimension and the complexity of our problem. Thus, we scaffold this methodology iteratively for expert agents to 'prime' the language model to respond with more accurate and usable metrics. Our approach guides the knowledge-generation process with meta-prompting techniques that do not necessarily relate to the original question, only within context. We also integrated databases where expert agents can consume direct information. This iterative prompt priming process is guided through conversations, denoted as the first circle in Fig 2. Since it is a conversation, the entire text from the previous stage is appended to the latter stage and passed to the language model. We detail them as three stages below:

Stage 1. The goal of stage 1 is to extract specific knowledge about the expert(s), absent information about the goal itself. The prompt requests detailed information regarding the experts' publications and insights into the field, together with a biography of the expert. This prompt-generated content refers to the expert's bibliography and describes the contributions made by the expert. One can easily imagine using a knowledge base or search tool to introduce specific content in this step that the LFM may not have access to, for example when including internal experts and their unique and specific expertise.

```
You are now consulting with {{expert}} in {{expert_field}}. Who is {{expert}}?
Provide me with a bio of this expert.
Then, discuss with {{expert}} on the following questions and more about {{expert}}:

  - What important publications and articles has {{expert}} written?
  - What are the insights from these publications and articles?

This description should be extremely elaborate and detailed.
It should cover all the important aspects of {{expert}}'s work.
For each contribution, it should be a standalone paragraph that describes the contribution
and insights.
```

Stage 2. The goal of stage 2 is to prune unnecessary information generated in stage 1 by exposing the goal rather than a direct transformation for the goal. This selection process aims to prevent unrealistic or impractical proposals.

```
During this consultation with {{expert}}, you are now tasked to {{goal}} for team {{teamX}}.
Based on {{expert}}'s expertise, What do you think are important elements to consider
when looking to {{goal}} for a team {{teamX}}? List out at least {{num_of_tools}} elements.
```

Stage 3. The goal of stage 3 is to transform the selected knowledge generated in stage 2 into metrics designed for this goal. This happens by exposing the kind of data available to the agent. The prompt gives further goal details and asks for proxies based on the previous elicitation with additional guidelines. The introduction of data scientists abstracts away the complexity of the system while creating function calls that are reasonable and quantitative with constraints. However, during this process, we do not ask for specifics regarding the implementation.

~~~
These elements make a lot of sense. Now, we need to consider the data we have.
{{db_description}}. You can use the table names to infer reasonable columns
that are stored in the database with common knowledge. The list of table names
are provided here: {{db_tables}}. Based on the consultation above, discuss with {{expert}}
to find reasonable metrics that can be used to measure the elements you have listed out.
For each of the elements, you should try to come up with mutually exclusive metrics
that can be used to measure the elements. Remember, the expert's insights would not be helpful
if we cannot calculate a good metric to serve as a proxy using our dataset in the database.
[cropped for brevity]
```
<metric>
    <metric_name></metric_name>
~~~

~~~
        <metric_description></metric_description>
        <metric_reason></metric_reason>
    </metric>
    ```
~~~

We call this process iterative prompt priming because of the intentionally gradual exposure of the agent to the goal itself, guiding the model to process tokens based on the response from the previous stage. This unique approach directs the language model to generate content based on the expert's information. The subsequent steps guide the text exchange between the GEMS and the 'response' to better align with the content. During our design process, we found that guiding the LFM through seemingly out-of-context content toward the actual goal elicits better responses from individual expert agents than merely providing an expert's name or a direct inquiry. Although we lack direct statistical analysis to support this design choice, from a Natural Language Processing (NLP) perspective, we are intentionally weighting specific conditions for token generation in the latter process. This design aligns with neuroscientific concepts, specifically predictive coding and synaptic plasticity. Predictive coding, as first published by [26], highlighted that the brain anticipates sensory inputs based on prior experiences, enabling vision neurons to process information more efficiently. Rather than asking an LFM-powered agent to provide information after being faced with a specific information elicitation task, the system leveraged this priming technique to establish the expert agent with prior knowledge in light of the complex problem. Synaptic plasticity refers to neural connections strengthening or weakening over time-based on activity levels, leading to more efficient learning and memory processes [14]. The GEMS mirrors this process by selecting and engaging with a handful of expert agents, where each agent, primed with specific knowledge, represents a different activated neuron simulating the synaptic plasticity framework.

3.2.2 Multiple Expert Agents. GEMS engages multiple independent expert agents with various perspectives to tackle a given goal. For example, team productivity can involve fields like organizational psychology, organizational behavior, and communication. As shown in Fig 1, experts come in from different perspectives to produce a comprehensive response. A panel of experts is chosen to execute the algorithm described above.

This design aims to support the generalization of responses, which was motivated by the tree-of-thoughts [31] approach. Although this implementation differs by using multiple experts following the same algorithm, rather than allowing for different reasoning processes, it allows diverse opinions, supporting a depth-first or breadth-first approach.

3.2.3 Judges. Judges incorporate human knowledge in designing decision-making mechanisms. Mechanism design influences the decision-making process and incentivizes different final results. For example, a plurality electoral rule tends to maintain dual-partisanship [28], while quadratic voting mechanisms aim to reduce the tyranny of the majority [25].

GEMS allows users to specify at a high level how the ranking results should be aggregated without the need to know the specific aggregation mechanism. For instance, the user might want to request an 'easy to understand and fair aggregation' which might cause GEMS to use the approval voting mechanism [5]. The judge uses a ReAct framework [32] which explains how it selected a specific decision-making mechanism to reduce errors.

Once this mechanism is selected, the judge uses the algorithm of this mechanism in conjunction with the ranking results from all expert agents to derive the final decision.

4 METHODS

We created 10 different goals and generated metrics using GEMS and the OpenAI GPT-4 API without any prompt engineering. We conducted a qualitative comparative study to evaluate the metrics generated by GEMS.

Goals. We used ChatGPT-4 to generate five common challenges software development teams face. These topics were reviewed by co-authors on this paper. Given that this GEMS implementation was designed to support software engineering teams, we rewrote these challenges into goals that a software engineering manager might want to achieve. Understanding that not all goals could be easily expressed [13], we created abstract goals that transformed detailed descriptions into less specific requirements a software engineering manager might set out to achieve. We referred to those as *complex goals* vs. *abstract goals*. Here are two sets of examples:

- Challenge: **Ensure Performance and Scalability**
  - Complex Goal: Design and implement a scalable architecture that supports dynamic scaling and optimizes performance under varying load conditions.
  - Abstract Goal: Build systems that grow with our success, handling more users effortlessly.
- Challenge: **Enhance Cross-Team Coordination**
  - Complex Goal: Establish a cross-functional liaison role responsible for coordinating activities and dependencies between teams, ensuring seamless integration and timely project progression.
  - Abstract Goal: Create seamless collaboration across all teams to work as one unified force.

Evaluation Process. As GEMS allows the user to specify a series of parameters, we first describe the parameters used for this evaluation. We asked GEMS to generate metrics to achieve a certain goal by using the following parameters:

(1) the number of perspectives or fields of study to consider when coming up with experts (i.e., 4)
(2) the number of experts to be selected for the panel of experts (i.e., 3)
(3) the expert selection criteria for the expert panel (i.e.,“"panel must contain 2 experts from the same field and 1 from a different field”)
(4) the number of metrics each expert will generate for use in the team matching process (i.e., 3).

More concretely, each time GEMS executed a task prompt based on the settings described above, the system generated 4 fields of study relevant to the specified task, then generated 3 experts for each field, as shown in Figure 1. For example for a field of study like development operations, 3 real-world experts are generated: one known for a book about IT transformation, another for pioneering continuous delivery, and the third for impactful research on DevOps practices. A total of 12 experts are generated, out of which 3 are selected for the expert panel that will then generate 9 metrics total to be used for team ranking and final team matching. These ranking results are then aggregated to provide a final team match outcome.

For comparison, we used OpenAI GPT-4 API to pass the same goals without specific prompt engineering. We call this the Vanilla approach, following works like Liu et al. [21]. LangChain [1] was used as a templating engine to generate the 9 different metrics for the Vanilla approach. We show the given prompt below:

```
I am a software manager and I want my team to [{goal}]. I want to locate a different team
↪  in the company that can help me achieve this goal. What are metrics that I should
↪  look for, assuming I only have access to their GitHub repository. and assuming I have
↪  a data scientist that can help me craft specific queries for these metrics:
```
[{{'function_name': 'function description'}}, ...]
```
```

Both systems used the OpenAI GPT-4 API (version gpt-4-0613). It is important to note that not all cases passed to the Vanilla version returned 9 metrics and descriptions. We kept the descriptions as is to preserve the authenticity of our results. We did not cherry-pick results; metrics were generated only once, unless there was a rate limit error. After the metrics were generated, the first author reviewed the metric name and description generated by both systems, extracted them into a spreadsheet for comparison, and annotated the differences. These differences were then coded into cases shown in the next section.

5 RESULTS

In this section, we highlight the differences observed in the metrics generated by both systems—GEMS vs. Vanilla. To illustrate key differences between systems, we summarize our findings and observations for each goal. Each of the 9 goals is denoted by a number, followed by `ABS` for abstract goal, or `CPX` for complex goal. We represent metrics generated by GEMS with G, and Vanilla with V. We reformatted the defined function names to `camelCase` for readability, and shared the descriptions generated as is for authenticity. Finally, we highlight observations from comparing the two systems with theoretical justifications for these differences.

5.1 GEMS Generates Complex Metrics Built Through Compositions

Metrics generated by GEMS often display greater complexity. Complexity here does not mean that the metric is more difficult to generate; instead, it indicates a composite function of direct metrics. In other words, it tries to compress different perspectives of measures into one measure. After matching the closest metrics from both systems, we show a few examples to highlight the difference. Table 1 presents three specific metric pairs across three different goals.

Observations: In the examples shown in Table 1, GEMS generated more comprehensive and detailed metrics compared to Vanilla. The GEMS metrics are more complex, as these metrics require mathematical operations or aggregation through composition. For example, `userEngagement` is a composition of various forms of user interaction over time using different actions. In contrast, the Vanilla metric counts only the number of contributors without considering their level of activity or engagement. The GEMS metrics `rateOfChangeAdoption` and `trackAdaptivePlanning` demonstrate the multiple dimensions that GEMS elicits compared to the Vanilla approach.

5.2 GEMS Generates Metrics With Specificity Grounded In Theories

Metrics generated by GEMS often display specificity grounded in theoretical frameworks. Here, specificity does not indicate that the metric is more limited in scope; instead, it highlights a clear and focused measure grounded in established theories due to iterative prompt priming. Let us examine two examples of specificity in Table 2.

Table 1. Metric complexity across systems

Goal 02 CPX: Implement a structured, cross-platform communication system enabling real-time collaboration and knowledge sharing among all software development teams.	
G `userEngagement`	The number of comments, reactions, and reviews on issues and pull requests from each user over a defined time frame.
V `numberOfContributors`	The total number of contributors to the repository.
Goal 01 ABS Adapt to market change, stay ahead of the curve, always ready to embrace change for our advantage.	
G `rateOfChangeAdoption`	Measuring the frequency and volume of code changes or additions that have been accepted and merged into the base branch.
V `commitFrequency`	The frequency of commits.
Goal 05 ABS Keep projects focused and on track, delivering what we promised on time.	
G `trackAdaptivePlanning`	Identifies instances of reassignment of issues, changes in issue labels, or changes in milestone dates and computes the average frequency of these changes.
V `issueResolutionTime`	The time taken to resolve issues.

Table 2. Metric specificity across systems

Goal 01 CPX Implement an agile response strategy that allows the team to quickly pivot and adapt to emerging technologies and market demands.	
G `refactoringFrequency`	This metric measures how frequently code refactoring is performed by examining the commit history for keywords related to refactoring or significant code alterations.[. . .]
V `codeChurn`	This function will measure the amount of code that is changed, added, or deleted over time. Agile teams should have low code churn rate as they aim for iterative and incremental development.
Goal 08 ABS Build a team culture that celebrates success and fosters strong bonds among team members.	
G `recognitionRatio`	[. . .]the frequency of appreciative or congratulatory comments made by a user. This could be coupled with other positive emojis or phrases used to recognize others' work. [. . .]
V `frequencyOfCommentsOnCommits`	Frequent comments can suggest a high level of collaboration and communication among team members.

Observations: In both examples, metrics are grounded in existing theories. In the first example, GEMS elected a renowned software developer, author, and speaker, recognized for contributions to software development practices and methodologies, as an expert to consult with. The corresponding metric that GEMS generated, `refactoringFrequency`, directly aligns with the importance of constant refactoring to support the software engineering team in adopting changes, as shown in prior research [9]. Unlike the Vanilla metric `codeChurn`, which tracks overall code updates, this metric specifically measures refactoring activities.

In the second example, GEMS was primed on an expert who brought emotional intelligence into the mainstream and integrated these concepts into leadership practices. Thus, unlike the Vanilla metric, GEMS looks for specific cues such as '+1', 'heart', 'hooray', 'laugh', or 'rocket' in the comment as a proxy which aligns with emotional intelligence theories [10].

Table 3. Metrics operationalization across systems

Goal 07 CPX Form a technology review board that evaluates and recommends technologies based on current and future project needs, considering factors like scalability, maintainability, and team expertise.

G `calculateEconomicImpact`	...calculate the 'economic impact' as the implementation cost minus the estimated revenue (ROI) that the technology would generate. The implementation cost can be inferred from (the) 'investment data' table, and the ROI could be estimated based on past performance data for similar technologies under 'repository history' and 'technology usage' records.
V `N/A`	Nothing comparable or of similarity

Goal 09 ABS Become the best place to work for software developers by offering growth and stability.

G `measureAutonomy`	...the autonomy level of a software developer ... include factors such as how often they're allowed to choose their own tools, set their own deadlines, or decide their approach to problem-solving. ...
V `N/A`	Nothing comparable or of similarity

5.3 GEMS Generates Metrics Harder to Operationalize and With More Assumptions

GEMS metrics offer valuable, holistic insights, but also bring challenges. First, the metric complexity and specificity can make them harder to operationalize and implement automatically. Second, some metrics come with specific assumptions that are not apparent from their descriptions. We illustrate these differences by comparing Vanilla and GEMS metrics in Table 3.

Observations: In Table 3, GEMS generated two metrics that reflect concepts stemming from existing literature which Vanilla did not generate. However, both metrics cannot be derived solely based on code repository information or readily available data. While the descriptions pointed to possible proxies that might be able to estimate these values, GEMS made assumptions that information such as the implementation cost of a technology is available to the system, or to the individuals operating the tool. This makes these metrics difficult to scale and make use of. That being said, these metrics are the first steps for teams to locate reasonable measures as proxies that reflect them.

5.4 GEMS Generates More Diverse Metrics

In spite of the occasional metric that is hard to interpret and operationalize, the metrics generated by GEMS exhibit greater diversity. The Vanilla system tends to repeat or use similar metrics, despite the wide range of goals specified. For example, across the 178 metrics generated by the Vanilla system for 10 cases, the top 5 frequently identified metrics are: `issueResolutionTime` appearing 14 times, `commitFrequency` appearing 13 times, `testCoverage` appearing 12 times, `codeChurn` appearing 11 times, `contributorsCount` appearing 11 times. These 5 metrics account for 34% of all 178 metrics.

On the other hand, GEMS generated 180 metrics with the following top 5 frequently identified metrics: `calculateAverageIssueResolutionTime` appearing 5 times, `communicationFrequency` appearing 4 times, `measureTestCoverage` appearing 3 times, `calculateCollaborationScore` appearing 3 times, and `alignmentScore` appearing 3 times. These metrics account for only 10.11% percent of all 180 metrics.

Table 4. Goal 05 ABS: Keep projects focused and on-track, delivering what we promised on time. </> denotes metrics that were successfully automatically implemented.

(a) Metrics generated by GEMS

G	Metrics generated by GEMS
</> `Calculate-IncrementalDeliveryRate`	[…] the number of closed issues or merged pull requests over a specific time period […] timestamp and statuses of issues and pull requests to measure this.
</> `Compute-Communication-Efficiency`	[…] average time it takes from the moment a new issue is created, until the first comment is made or action is taken. It will infer this information from the timestamp of issue creation and the timestamp of first interaction (comment or action).
</> `Track-Adaptive-Planning`	[…] how quickly and frequently the project's plans or priorities are adapted in response to new developments or changes. It identifies instances of reassignment of issues, changes in issue labels, or changes in milestone dates, and computes the average frequency of these changes.
</> `Feature-Complexity-Index`	[…] average number of features (or tasks) per release (or milestone). This calculation will require access to the 'milestone' and 'issue' tables to count the number of issues completed per milestone. Additionally, dependencies and relations between issues can also be used to infer complexity.
</> `Mean-Time-To-Resolve`	[…] the average time taken to close an issue once it is opened. It will need data from the 'issue' table to calculate the time difference between the creation and closing of each issue.
</> `Task-Completion-Rate`	[…] ratio of tasks or issues that are closed to the total number of tasks. This calculation requires access to the 'issues' table to obtain data about created and closed issues.
</> `Calculate-Average-Issue-Resolution-Time`	[…] average time taken to resolve issues. It would involve fetching data from tables storing issue creation and closing timestamps and calculating the difference in days.
</> `Calculate-Backlog-Management-Efficiency`	[…] the time taken (in days) between adding an issue to the backlog and that issue being closed. It would involve extracting data from the tables that store issue details and timestamps of when issues were created and closed.
</> `Calculate-Sprint-Completion-Rate`	[…] the percentage of tasks (issues and pull requests) from the backlog that were completed in a sprint, inferred by open and closed timestamps and commit information. It would require data from the tables that store issue, pull request and commit details, appropriately joined and filtered by time frames.

(b) Metrics generated by Vanilla

V	Metrics generated by Vanilla
`Bug-Fixing-Rate`	[…] the rate at which bugs are fixed within the repository. A high bug fixing rate signifies a team that is effective at identifying and addressing issues.
`Code-Churn`	[…] amount of code that gets modified or deleted after being checked in, which is an indicator of code stability and can suggest the effectiveness of the team.
`Code-Quality-Metrics`	[…] quality of the code in the repository. High-quality code indicates a team that values accuracy and efficiency.
`Code-Review-Time`	[…] time taken to review and merge code. A shorter review time suggests a team that can efficiently manage its workload.
`Commit-Frequency`	[…] frequency and regularity of commits. A team that regularly contributes to their repository is likely to be more focused and able to deliver on time.
`Contribution-Diversity`	[…] diversity of contributions within the repository. A diverse range of contributions suggests a team with a wide range of skills and expertise.
`Issue-Resolution-Time`	[…] the time taken to resolve issues. A shorter resolution time indicates a more efficient team.
`Project-Completion-Rate`	[…] rate at which projects are completed within the repository. A high project completion rate indicates a team that can deliver what it promises on time.
`Pull-Request-Rate`	[…] rate at which pull requests are created and closed. A high pull request rate may signify an active team.

Table 5. Goal 05 CPX: Implement a rigorous project management protocol that requires detailed documentation and approval for all changes to project scope or objectives. </> denotes metrics that were successfully automatically implemented.

(a) Metrics generated by GEMS

G	Metrics generated by GEMS
`calculate-Risk-Identification-And-Plan`	[...] identification of risks and their mitigation planning related to the changes in the project scope. This can be done by analyzing the issue tracking data and corresponding mitigation steps taken, and whether the issues were associated with scope changes in the git commits or pull requests.
`calculate-Scope-Change-Frequency`	[...] number of times the scope/objective of the project changes. [...] counting the number of times significant modifications are made to the git repository, indicated by either major commits or pull requests that significantly alters the codebase or its documentation.
</> `measure-Team-Involvement-In-Changes`	[...] involvement of team members in making changes to the project scope or objectives. [...] tallying the number of unique individuals contributing to commits or pull requests associated with significant changes, along with the frequency of their contributions.
`impact-Analysis-Score`	[...] calculated by assessing the proportion of proposed changes that were backed by an impact analysis report. An impact analysis report documents the anticipated effects of the proposed change on other aspects of the project.
`process-Adherence-Score`	[...] comparing the number of changes made in a project that followed the documented process against the total number of changes made. [...] whether the proposal, documentation, and approval steps were correctly undertaken for every change made.
</> `team-Communication-Score`	[...] analyzing the number of documented communications (like discussions, meetings records, or emails) related to a proposed change against the total number of changes made. A higher ratio indicates a more robust communication framework in place.
</> `change-Approval-Rate`	[...] proportion of proposed changes to the project scope or objectives that went through [...] rigorous project management protocol. This can be measured by checking if a formal approval record exists for each proposed change.
`documentation-Compliance-Score`	[...] calculates a score based on the completeness and timeliness of the documentation updates. Completeness can be gauged by checking if all required sections of the documentation are filled out. Timeliness can be checked by seeing if the documentation update was done within a pre-specified time [...] (like change approval).
`stakeholder-Involvement-Index`	[...] level of stakeholder involvement in decision-making processes. [...] tracking the frequency and degree of engagement of stakeholders in important discussions, their involvement in the review and approval of proposed changes, and their presence in meetings.

(b) Metrics generated by Vanilla GPT4API

V	Metrics generated by Vanilla
`code-Reviews`	The presence of code reviews shows a commitment to quality assurance, a key component of rigorous project management.
`commit-Messages`	Analyzing commit messages can provide insights into how well the team documents their changes. Look for detailed, meaningful messages.
`issues-Created-And-Closed`	This can show how well the team manages their project's scope and tracks bugs and desired features.
`merge-Conflicts`	A low number of unresolved merge conflicts may indicate a team that effectively manages their workflow.
`number-Of-Branches`	Frequent use of branches may indicate that the team is using a feature branching workflow, often associated with rigorous project management protocols.
`number-Of-Commits`	This will show you how active the team is. A team that's well-versed in project management should have a consistent number of commits.
`number-Of-Contributors`	A higher number of contributors may indicate a larger or more collaborative team.
`pull-Requests`	The number of pull requests, their frequency, and the quality of their descriptions can also indicate how rigorously the team manages their work.
`release-Frequency`	The frequency and regularity of releases could indicate a well-structured development process.

5.5 Detailed Case Analysis

In the previous subsection, we discussed characteristics observed across all the metrics generated for the 10 cases by selecting pairs of metrics between the two systems. In this subsection, we focus on two specific goals and describe these differences holistically. We selected *Goal 05*, relating to the challenge of "Prevent Scope Creep" that many software developer teams face. The two derived goals are listed as:

- Goal 05 ABS: Keep projects focused and on-track, delivering what we promised on time.
- Goal 05 CPX: Implement a rigorous project management protocol that requires detailed documentation and approval for all changes to project scope or objectives.

We select *Goal 05* for the detailed case analysis because the programmer agent in GEMS was able to automatically generate the code implementation (using AutoGen) for all the metrics proposed by GEMS for *Goal 05 ABS*. Thus, for this goal we show a fully functional, end-to-end process of metrics being generated, implemented, and executed to find a team match that achieves the given goal. The other goals that did not have code implementations for all metrics successfully auto-generated were not marked as failures since automatic code generation is not the primary goal of this paper. We limit the scope of this project to metric generation because code generation itself is a separate, growing field of study [3, 15, 30]. This example provides a baseline benchmark to compare against for future implementations, as more advanced self-coding systems can only enhance GEMS performance. Table 4 and Table 5 list results for *Goal 05 ABS* and *Goal 05 CPX*, respectively. We placed a </> for the metrics where code implementation was auto-generated by AutoGen, and then executed by GEMS by using software communities data from the database. For example, for the first metric in Table 5a, AutoGen generated Python and SQL code that implements `CalculateIncrementalDeliveryRate` by executing SQL code querying the database for the number of closed issues in 2022.

When comparing metrics generated by GEMS and the Vanilla approach, as shown in Table 4, we see that they echo the characteristics discussed in the previous subsection. When we consider the 9 metrics GEMS generated, we see that some metrics can be similar in terms of the attributes they consider. For instance, GEMS generated `MeanTimeToResolve` and `ComputeCommunicationEfficiency`. The former calculates the average time it takes for an issue to be resolved. The latter considers the average time an issue receives an interaction. As such, both metrics implicitly prioritize and weight specific measures more, in this case, time and issues, by creating variations within all 9 metrics that make use of these same measures.

Other times, composite metrics that GEMS generated contain measures that other metrics cover. In this example, `calculateSprintCompletionRate` partially covers the ratio of open to closed issues which `taskCompletionRate` aims to measure. This overlap became an implicit weighting of specific measures when the final judge considered all 9 metrics.

In contrast with GEMS metrics, metrics generated by the Vanilla approach are broad and encompass a wide variety of items that do not necessarily address the given goal. The metrics typically represent general software engineering concepts without being tailored specifically to the given task.

Next, we examine how metrics change when goals become more specific. Shifting from abstract to concrete, we compare the metrics listed in Table 5. First, we notice the sensitivity in metrics generated by GEMS in this case. Adding concepts such as "project management protocol" and "detailed documentation" to the goal specification led to the increase in metrics that are less technical, such as `ProcessAdherenceScore` and `calculateRiskIdentificationAndPlan`, as well as specific measures such as `documentationComplianceScore`. While GEMS did not automatically generate the corresponding code for these metrics, implementations for them could be crafted with human intervention, or using more advanced data analysis tools. These metrics, on the other hand, highlight better metrics that can contribute to realizing the given goal. On the other hand, when we look at the vanilla metrics generated for the complex goal, the metrics generated remain low-level snapshots of repository information. Furthermore, many of these metrics are very similar to the metrics generated using the abstract goal.

In summary, when considering Goal 05, we showed that GEMS generated 9 metrics that implicitly weighted specific measures that it believed were critical to improving the given goal. When comparing metrics between the abstract and complex goals, we observe that GEMS generated metrics are more sensitive to the change and tailored the responses by using specific items grounded in theories.

6 DISCUSSION

6.1 GEMS as a Copilot

GEMS acts as a copilot in constructing proxies tailored to the specified goal, and using the data at hand. As noted in the Results section, the system sometimes generates metrics that are overly narrow, or overfits its conclusions. As such GEMS plays the role of the real-world experts who bring **theoretical insights** to the given problem, and local individuals familiar with the **context of the problem** would guide the experts to apply these insights effectively.

The literature on copilot roles and expert-novice collaboration supports this. Constructing knowledge requires active participation, developed and used through engagement with the activity, context, and culture [20]. GEMS lowers the barriers by surfacing expert knowledge directly to individuals without actual experts present. In the example, the metric `calculateEconomicImpact` shown in Table 3 encouraged using the ROI of previous technology usages as a proxy for evaluating technologies out of thousands, if not millions, of possible metrics. As models improve with advancements in code generation and logical reasoning, they will be able to offer better, more specific recommendations. However, individuals familiar with the problem context will still need to guide metrics, positioning GEMS as a Copilot rather than a people replacement.

6.2 Mapping to Expert Decisions

One of the strengths of the metrics generated by GEMS is its composition and specificity; akin to the work of real-world experts. For instance, experts often already had a sense of the specific relationships, concepts, and thresholds among a selection of measurements. GEMS takes a similar approach where measurements then form metrics when a scenario is given.

Herbert Simon's work on intuition and judgment posits that intuition and sound judgments are outcomes of extensive practice and experience, which he denotes as "frozen analysis" [29]. These internalized analyses allow experts to respond rapidly to familiar patterns when given a context. In many cases, experts rely on experience-based intuition and pattern recognition in dynamic environments [17], thereby reducing errors [12]. These findings support the use of the iterative prompt priming technique that GEMS employs.

By making pre-processed information (in this case an expert's prior works) available to the Large Foundation Model (LFM), the model does not have to regenerate or digest information from scratch, allowing the model to generate more accurate and expert-like decisions. Hence, we observe the different metrics generated by the Vanilla model and the GEMS system.

Recall the metrics grounded in existing theories listed in Sec. 5.2. If GEMS was able to map critical works in the field of experts who have unique, significant experiences to specific metrics, it is reasonable to project its capability when users within a software company provide GEMS with information available internally only. This information can be knowledge bases, personas, or even experts who do not have an external facing profile, but have significant know-how or skill sets within the organization. In these settings, GEMS would be able elect and translate these characteristics into concepts that inform context-dependent metrics. This tool has the potential to transform how individuals within organization solicit the unique expertise of their colleagues and evolve their approach to collaboration.

6.3 Blueprinting Metrics to Solve Difficult Problems using LFMs

GEMS, in its role as a copilot, demonstrates its ability to forge expert knowledge into context-specific solutions, effectively crafting blueprints for solving difficult problems by leveraging the capabilities of Large Foundation Models (LFMs).

Prior research highlights two primary challenges when it comes to crafting metrics. The first challenge lies in initiating metrics that can effectively measure and resolve issues [11, 22]. Once this challenge is overcome, teams may still misuse inappropriate metrics for their goals. For example, in software engineering, developers frequently equate productivity with lines of code, a simplistic measure that fails to capture the true complexity of team output. Experts have corrected such measurements by developing metrics that better reflect a team's collective contributions, encompassing multiple dimensions of developer work [8]. In these situations, teams typically require expert guidance to navigate and select metrics *suitable* for comparison.

GEMS positions itself as a blueprinting tool, guiding the creation of a more comprehensive set of metrics. These blueprints are designed to stimulate informed discourse among teams, leading to better comparisons and decisions without an immediate need for physical experts, particularly when teams face complex and challenging problems.

The iterative prompt priming technique with the expert panel selection design translates complex problems into multiple perspectives. Thus, even if GEMS introduces complex and less operationalized metrics, it prevents novices from selecting more accessible but possibly poor metrics. Instead, it seeds a conversation that overcomes this hurdle, allowing local expertise and discussion to facilitate more situated metrics based on GEMS's initial recommendations. This design also allows CMS to generate metrics covering both breadth and depth. The case study in the results section (Sec. 5.5) demonstrates the breadth of metrics generated, showcasing the system's ability to identify experts from diverse domains. While priming in behavioral economics can introduce biases, intentionally priming models are used here to reduce overly narrow metrics and solutions.

6.4 Limitations and Future Work

While GEMS metrics show that such systems can generate *better* metrics given a goal and a dataset, we do not know how individual users would use such a system. Our report did not evaluate how decision-makers use team pairings generated by GEMS. The system still relies on users providing authentic and trustworthy information. Correctness and transparency in software version control are crucial for accurate pairing suggestions. Decision-makers must take responsibility for final decisions and effectively use systems like GEMS. Thus, future work should experiment with managers and decision-makers to understand how they evaluate the metrics generated and make use of systems like GEMS.

7 CONCLUSION

Identifying suitable metrics to address challenging goals in improving software engineering practices is inherently difficult. Traditionally, individuals have often relied on suboptimal metrics that are more readily available. While prior research suggests the possibility of learning tacit knowledge from contextually similar teams, it still necessitates the design of measures that can effectively identify these teams.

In this technical report, we proposed GEMS as a prototype applied to OSS community data powered by Large Foundation Models (LFMs) to assist individuals in designing more suitable metrics for specific software practice challenges. We introduced the iterative prompt priming technique and designed a multi-agent expert panel with a judge in our implementation. Our results highlight the differences between the metrics generated by GEMS and those produced by a Vanilla question-and-answer approach when addressing a given goal, illustrating the potential impact of GEMS in a software engineering industry.

Through our case analyses, we demonstrated that GEMS generates complex metrics grounded in theories, making them more specific to the given problem. Additionally, GEMS produces a more diverse set of metrics that adhere more closely to the specified goals. Finally, we showed that GEMS-generated metrics are more sensitive to the nuances of specific goals.

Our findings have significant implications for how LFMs can be utilized to obtain expert perspectives, particularly when direct access to experts is limited. We envision future work that further investigates software engineering managers' and decision-makers' perceptions when provided with metrics generated by GEMS.

ACKNOWLEDGMENTS

We would like to thank our Microsoft colleagues, interns, and partners who provided feedback that helped guide the direction of this work. Ti-Chung Cheng conducted this work as a research intern in Microsoft Research's Software Analysis and Intelligence in Engineering Systems (SAINTES) Group.

REFERENCES

[1] [n. d.]. LangChain. https://www.langchain.com/.
[2] 2024. Guidance-Ai/Guidance. guidance-ai.
[3] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting Is Programming: A Query Language for Large Language Models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1946–1969.
[4] Huaimin Wang Bogdan Vasilescu, Yue Yu. [n. d.]. Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *ESEC/FSE 2015: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.*
[5] Steven J. Brams and Peter C. Fishburn. 2007. *Approval Voting* (2nd ed ed.). Springer, New York.
[6] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early Experiments with GPT-4. arXiv:2303.12712 [cs]
[7] Margaret Elliott and Walt Scacchi. 2003. Free Software Developers as an Occupational Community: Resolving Conflicts and Fostering Collaboration. *ACM Siggroup Bulletin* 24 (Nov. 2003). https://doi.org/10.1145/1027232.1027237
[8] Nicole Forsgren, Margaret-Anne Storey, Chandra Maddila, Thomas Zimmermann, Brian Houck, and Jenna Butler. 2021. The SPACE of Developer Productivity: There's More to It than You Think. *Queue* 19, 1 (Feb. 2021), 20–48. https://doi.org/10.1145/3454122.3454124
[9] Martin Fowler. 2019. *Refactoring: Improving the Design of Existing Code* (second edition ed.). Addison-Wesley, Boston.
[10] Daniel Goleman. 1998. *Working with Emotional Intelligence.* Bantam.
[11] Dina Gray, Pietro Micheli, and Andrey Pavlov (Eds.). 2014. *Measurement Madness: Recognizing and Avoiding the Pitfalls of Performance Management.* Wiley, Chichester, West Sussex.
[12] Kenneth R Hammond, Robert M Hamm, Janet Grassia, and Tamra Pearson. 1987. Direct Comparison of the Efficacy of Intuitive and Analytical Cognition in Expert Judgment. *IEEE Transactions on systems, man, and cybernetics* 17, 5 (1987), 753–770.
[13] Edie L Holcomb. 2001. *Asking the Right Questions: Techniques for Collaboration and School Change.* Corwin Press.
[14] E. R. Kandel. 2001. The Molecular Biology of Memory Storage: A Dialogue between Genes and Synapses. *Science* 294, 5544 (Nov. 2001), 1030–1038. https://doi.org/10.1126/science.1067020
[15] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural Language to SQL: Where Are We Today? *Proceedings of the VLDB Endowment* 13, 10 (2020), 1737–1750.

[16] Ghadeer A Kintab, Chanchal K Roy, and Gordon I McCalla. [n. d.]. Recommending Software Experts Using Code Similarity and Social Heuristics. ([n. d.]).

[17] Gary A. Klein, Judith Orasanu, Roberta Calderwood, and Caroline E. Zsambok (Eds.). 1993. *Decision Making in Action: Models and Methods.* Ablex Publishing, Westport, CT, US. xi, 480 pages.

[18] Amy J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, Minneapolis, MN, USA, 344–353. https://doi.org/10.1109/ICSE.2007.45

[19] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-Answer Questions about Code. In *Evaluation and Usability of Programming Languages and Tools*. ACM, Reno Nevada, 1–6. https://doi.org/10.1145/1937117.1937125

[20] Jean Lave. 1991. Situating Learning in Communities of Practice. In *Perspectives on Socially Shared Cognition*. American Psychological Association, Washington, DC, US, 63–82. https://doi.org/10.1037/10096-003

[21] Jiacheng Liu, Alisa Liu, Ximing Lu, Sean Welleck, Peter West, Ronan Le Bras, Yejin Choi, and Hannaneh Hajishirzi. 2022. Generated Knowledge Prompting for Commonsense Reasoning. arXiv:2110.08387 [cs]

[22] Yaoli Mao, Dakuo Wang, Michael Muller, Kush R. Varshney, Ioana Baldini, Casey Dugan, and Aleksandra Mojsilović. 2019. How Data ScientistsWork Together With Domain Experts in Scientific Collaborations: To Find The Right Answer Or To Ask The Right Question? *Proc. ACM Hum.-Comput. Interact.* 3, GROUP (Dec. 2019), 237:1–237:23. https://doi.org/10.1145/3361118

[23] David W. McDonald and Mark S. Ackerman. 1998. Just Talk to Me: A Field Study of Expertise Location. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*. ACM, Seattle Washington USA, 315–324. https://doi.org/10.1145/289444.289506

[24] Christopher Poile, Andrew Begel, Nachiappan Nagappan, and Lucas Layman. [n. d.]. Coordination in Large-Scale Software Development: Helpful and Unhelpful Behaviors. ([n. d.]).

[25] Eric A Posner and E Glen Weyl. 2018. *Radical Markets: Uprooting Capitalism and Democracy for a Just Society.* Princeton University Press.

[26] Rajesh P. N. Rao and Dana H. Ballard. 1999. Predictive Coding in the Visual Cortex: A Functional Interpretation of Some Extra-Classical Receptive-Field Effects. *Nat Neurosci* 2, 1 (Jan. 1999), 79–87. https://doi.org/10.1038/4580

[27] Elaine M. Raybourn, Reed Milewicz, David M. Rogers, Benjamin Sims, Greg Watson, Elsa Gonsiorowski, and James Willenbring. 2021. *A Data-driven Approach to Rethinking Open Source Software Organizations as a Team of Teams.* Technical Report SAND2021-9378C. Sandia National Lab. (SNL-NM), Albuquerque, NM (United States).

[28] William H. Riker. 1982. The Two-party System and Duverger's Law: An Essay on the History of Political Science. *American Political Science Review* 76, 4 (Dec. 1982), 753–766. https://doi.org/10.2307/1962968

[29] Herbert Alexander Simon. 1997. *Models of Bounded Rationality: Empirically Grounded Economic Reason.* Vol. 3. MIT press.

[30] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2023. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155 [cs]

[31] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. https://doi.org/10.48550/arXiv.2305.10601 arXiv:2305.10601 [cs]

[32] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs]

[33] Svetlana Yarosh, Tara Matthews, and Michelle Zhou. 2012. Asking the Right Person: Supporting Expertise Selection in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12)*. Association for Computing Machinery, New York, NY, USA, 2247–2256. https://doi.org/10.1145/2207676.2208382

A SUPPLEMENTARY DIAGRAMS

B EXPANDED REFERENCE CASES

We provide the remaining 4 cases in the appendix.

Fig. 3. System Architecture Diagram of GEMS

Table 6. Goal 01 ABS: Stay ahead of the curve, always ready to embrace change for our advantage. </> denotes metrics that were successfully automatically implemented.

(a) Metrics generated by GEMS

G	Metrics generated by GEMS
CountModern-MethodUsage	[…] the usage of modern software development methodologies in a repository. This can be measured by looking at the commit messages for references to methodologies such as Agile, DevOps, or other practices. A higher count suggests a more modern approach to software development.
</> Measure-Learning-Adaptation	[…] the adoption of new technologies or methodologies over time. This might be inferred through the introductions of new libraries or tools in the repository, significant shifts in coding practices, or introductions of new types of issue labels. A team that quickly adopts new technologies is likely to be proactive in staying current with new developments.
</> Track-Innovation-Creativity	[…] the number and frequency of innovative, non-traditional solutions adopted in a repository. This can be evaluated by assessing the implementation of unique coding practices or pioneering technologies in a repository, as well as the number of forks or stars a repository gets. High numbers suggest a high degree of innovation and creativity.
</> frequencyOf-IssueOpened	[…] rate at which new issues are opened for a given repository in a given period. This can be calculated using the count of new issues created divided by the total time observed.
numberOfActive-Contributors	This metric measures the number of unique active contributors to a particular repository within a defined timeframe. This can be computed by tracking the unique users making commits.
</> rateOf-ChangeAdoption	[…] rate at which changes (like new features or bug fixes) are adopted in a particular repository. This can be calculated by measuring the frequency and volume of code changes or additions that have been accepted and merged to the base branch.
AgilePractice-AdoptionMetric	[…] frequency of commit histories and pull requests, the essential elements of Agile practices. It calculates the average number of commits and pull requests per week for each repository. Higher frequency suggests a more Agile and iterative code development practice.
</> Code-Refactoring-Metric	[…] frequency of code refactoring by counting the average number of code modifications per commit. Frequent modification indicates code optimization or refactoring exercises are taking place.
Innovation-Experimentation-Metric	[…] evaluates the extent of innovation and experimentation by monitoring the introduction of new tools and technologies in the repositories, inferred by the frequency of files with new extensions and the introduction of new topics in the repository.

(b) Metrics generated by Vanilla

V	Metrics generated by Vanilla
codeChurn	[…] the amount of code that is added and then removed or changed. Lower code churn can indicate stable, mature codebase while higher code churn may indicate active development and willingness to experiment with changes.
codeReviewTime	[…] the time taken to review and merge pull requests. A shorter review time can indicate efficiency and readiness to move forward with changes.
commit-Frequency	[…] the frequency of commits. Frequent commits can indicate the team's continuous effort to improve and adapt.
contributor-Count	[…] the number of contributors. A high number of contributors can indicate a diversified team that can bring in different perspectives for change.
dependency-Update-Frequency	[…] how often dependencies are updated. Regular updates indicate the team's readiness to adapt to new technologies and changes.
issuesOpened-ClosedRatio	[…] the ratio of opened to closed issues. A ratio close to 1 can indicate the team's ability to resolve issues and adapt to changes.
mergedPull-Requests	[…] the number of pull requests that have been merged. A high number of merges can indicate the team's ability to successfully implement changes.
pullRequest-Frequency	[…] the frequency of pull requests. High frequency can indicate that the team is continuously working on improving their codebase.
release-Frequency	[…] how often new releases are made. Frequent releases can indicate a team that's able to quickly respond to changes and deliver results.

Table 7. Goal 01 CPX: Implement an agile response strategy that allows the team to quickly pivot and adapt to emerging technologies and market demands. </> denotes metrics that were successfully automatically implemented.

(a) Metrics generated by GEMS

G	Metrics generated by GEMS
</> `Customer-Orientation-Index`	[. . .] the extent of customer orientation of a team, calculated by aggregating and analyzing comments, reactions, pull requests, and issues data. Higher indices indicate greater attention to user-reported issues and faster response times.
`Disruption-Adaptation-Index`	[. . .] the frequency and speed at which a team adopts and integrates emerging technologies, calculated by the average time gap between consecutive adoptions. Faster adaptation indicates less time between adoptions.
</> `Team-Learning-Index`	[. . .] the growth in technical repertoire of the team over time, assessed by the variety and complexity of libraries and practices in repositories and their change over time. A higher index suggests a culture of continuous learning.
`Calculate-ChangeAdaption-Rate`	[. . .] the ratio of project elements adjusted and implemented due to emerging technologies or market demands to total project elements in a given time frame.
`Calculate-Customer-Feedback-Implementation`	[. . .] the ratio of customer feedback points implemented to the total number of feedback points received in a given time frame.
`CalculateSprint-CompletionRate`	[. . .] the ratio of completed tasks to total set tasks at the beginning of every sprint cycle, with a completed task defined as one checked off as completed in the sprint.
</> `Adaptive-PlanningIndex`	[. . .] the frequency of team check-ins, code pushes, and code reviews, plus the number of changes made in response to external tech trends, determined by commit or issue descriptions.
</> `Integration-Delivery-Frequency`	[. . .] how often new updates, features, or patches are integrated and delivered within the project, determined by commit history and pull request data.
</> `Refactoring-Frequency`	[. . .] the frequency of code refactoring, examined through commit history for keywords related to refactoring or significant code alterations. Commit comments and changes provide additional insights.

(b) Metrics generated by Vanilla GPT4API

V	Metrics generated by Vanilla
`branchUsage-Frequency`	[. . .] measure how frequently the team uses branches. Agile teams often use branches for different features or tasks.
`codeChurn`	[. . .] measure the amount of code that is changed, added, or deleted over time. Agile teams should have low code churn rate as they aim for iterative and incremental development.
`codeReview-Process`	[. . .] measure the thoroughness of code reviews. Agile teams often emphasize on code reviews as part of their development process.
`commit-Frequency`	[. . .] measure the frequency of commits. Agile teams tend to commit changes often.
`contributors-Count`	[. . .] count the number of contributors. Agile teams usually have many contributors, indicating a high level of collaboration.
`issue-Resolution-Time`	[. . .] measure the average time taken to resolve issues. Agile teams should be able to resolve issues quickly.
`pullRequest-Acceptance-Rate`	[. . .] measure the acceptance rate of pull requests. High acceptance rate may indicate a well-coordinated team.
`technology-Stack-Diversity`	[. . .] measure the diversity of languages and technologies used in the repository. This could indicate the team's ability to adapt to different technologies.
`testCoverage`	[. . .] measure the extent of test coverage in the codebase. Agile teams usually emphasize on testing as part of their development process.

Table 8. Goal 02 ABS: Foster an environment where every team member feels connected and informed. </> denotes metrics that were successfully automatically implemented.

(a) Metrics generated by GEMS

G	Metrics generated by GEMS
Evaluate-Adaptability	[...] team's adaptability by determining how often and how effectively changes are incorporated into the projects. [...] analyze the frequency and nature of commits and pull requests along with the corresponding comments and reactions. [...] measure how quickly and smoothly new issues are created and resolved, and how new ideas, suggested in the form of pull requests or issues, are accepted and merged into the projects.
Measure-Communication-Flow	[...] level of interactions among team members by analyzing the frequency, variety, and response time of comments, reviews, and reactions in the repositories. [...] the number of comments and responses in pull requests, issues, and commit reviews, along with the number of positive reactions to these interactions.
</> Track-Collaborative-Efforts	[...] track the degree of collaborations of team members by calculating the average number of contributors per commit, per issue, and per pull request. [...] assignments of tasks to gather info on how workload is distributed among team members. [...] measure the activities of joint problem solving in issues and pull requests.
</> Calculate-Cultural-InclusivityScore	[...] calculated by analyzing the various types of GitHub interactions among team members captured in the data. This can include the number of comments made by each member, the diversity of projects they contribute to, and the receptivity of their contributions (measured through reactions, mentions, or replies).
</> Determine-CareerAnchor-Alignment	[...] the correlation between a team member's career anchors and their contributions on GitHub. Though not explicit in the database, career anchors can be inferred by looking at the tasks individuals choose, the complexity of tasks they handle, and their growth in area of contribution. [...]
</> Measure-Continuous-LearningGrowth	[...] quantifies the learning curve of each team member by tracking how their contributions evolve over time in terms of complexity (measured by lines of code), diversity (measured by project involvement), and acceptance (measured by merge rate for pull requests). Additionally, the number of issues raised and resolved, and the growth in the breadth of their engagement (number of different repositories contributed to) can be used as indicators of learning.
</> Alignment-Score	Measure the alignment of team members on project goals and tasks.[...] such as pull requests, issues resolved, and consistency in commit messages. This score will be higher if there is consistency in the activity revolving around the project goals.
</> Communication-Frequency	Measure the frequency of interactions between team members. [...] include comments, reactions, and reviews on a particular project. [...] obtained from tables such as those storing issue discussion or pull request discussion data. [...]
DiversityScore	Calculate a diversity score based on the variety of contributors in a project. [...] geographical location, affiliation, and role of each contributor, inferred by user metadata and association with or-

(b) Metrics generated by Vanilla

V	Metrics generated by Vanilla
Active-Discussions	Counts the number of active discussions on pull requests and issues. Active discussions can indicate good communication within the team.
CodeChurn	Measures the amount of code that is rewritten or deleted after being written. Higher churn might indicate a lack of clarity or agreement within the team.
CodeReview-Frequency	Estimates the frequency of code reviews. Regular code reviews indicate a team that shares knowledge and improves together.
CommitComment-Frequency	Counts the number of comments made on commits. Comments can be a sign of active discussion and feedback.
Commit-Frequency	Measures the frequency of commits. Regular commits indicate good collaboration and active engagement.
Contributors-Diversity	Assesses the diversity of contributors in terms of the number of different people making commits, opening issues, and submitting pull requests. A diverse range of contributors may suggest a more inclusive environment.
Issue-Resolution-Time	Measures the average time taken to resolve issues. Faster resolution times can suggest a well-coordinated team.
PullRequest-Frequency	Assesses the frequency of pull requests. Frequent pull requests can indicate a collaborative environment.
Repository-StarGazers	Counts the number of stars the repository has. This can be a sign of the team's impact within the organization.

Table 9. Goal 02 CPX: Implement a structured, cross-platform communication system enabling real-time collaboration and knowledge sharing among all software development teams. </> denotes metrics that were successfully automatically implemented.

(a) Metrics generated by GEMS

G	Metrics generated by GEMS
`CalculateDeviceAndOsCompatibility`	This function calculates the ratio of successfully completed activities (like commits, pull requests, etc.) to attempted activities across different devices and operating systems. This could be extracted through logs capturing activity metadata.
`CalculateInformationFindability`	This function calculates the average time taken by a user to find a piece of information or perform an action in the system. This could potentially be obtained from system logs capturing timestamped user behaviours.
`CalculateUserInteractions`	This function calculates the count of interactions a user has with the platform. It includes all activities like comments, commits, pull requests, issues raised, reviews, reactions etc. The metric can be calculated by summing up all actions performed by a user in the platform across all tables storing user information.
`AnalyzeObjectOrientedDesignAdoption`	This method should analyze the commit history and associated files to identify if object-oriented principles are adopted. It can be achieved by detecting signature practices of object-oriented programming such as encapsulation, inheritance, and polymorphism in the code and seeing patterns of their application in the development process.
`AssessCollaborationAndKnowledgeSharing`	This function should be designed to examine the user interactions data i.e., comments, reactions, review data, and assignees associations, to evaluate the level of collaboration and knowledge sharing happening. Milestones can be analyzed for cooperative planning, PR and issues comments for knowledge sharing, and user reactions for feedback loops. The degree of interconnectedness between teams and within teams can also be quantified.
`ComputeUnifiedModelingUtilization`	This function should scrutinize the historical record of commits and associated metadata files to determine the extent of UML. This can involve searching for specific file formats (.uml, .puml, .xmi, etc.) and evaluating the frequency and recency of their updates.
</> `AverageIssueResolutionTime`	This metric calculates the average time taken to resolve an issue. This can be calculated from the timestamp of when an issue is opened to when it is closed.
`ContinuousIntegrationFrequency`	This metric measures the frequency of integrations into the main branch done by the teams. This can be calculated by taking count of all the merged pull requests into the main branch over a defined period (monthly, quarterly, etc).
</> `UserEngagement`	This metric measures the level of user engagement in the project. User engagement can be quantified in terms of the number of comments, reactions, reviews on issues and pull requests from each user over a defined timeframe.

(b) Metrics generated by Vanilla

V	Metrics generated by Vanilla
`AverageResponseTimeToIssues`	This function will calculate the average response time to the issues raised. It would help you understand how quickly the team is able to address the issues.
`CodeChurn`	This function will calculate the rate at which code is added, modified, or deleted. It can help you understand the stability of the project.
`FrequencyOfCommits`	This function will calculate the frequency of commits in the repository. Frequent commits usually indicate an active development process and good collaboration among the team.
`NumberOfBranches`	This function will return the number of branches in the repository. Multiple branches often suggest that the team is capable of working on different features or tasks simultaneously.
`NumberOfContributors`	This function will return the total number of contributors to the repository. It will help understand the team size and its potential capacity.
`NumberOfForks`	This function will return the number of forks. It can give you an idea about the number of people or teams who are interested in their projects or using their code.
`NumberOfIssuesOpenedVsClosed`	This function will compare the number of opened issues versus closed ones. It will help you understand how effectively the team is solving the issues.
`NumberOfPullRequests`	This function will return the number of pull requests. This metric can be useful to understand the team's code review process and how often they merge their code.
`NumberOfStars`	This function will return the number of stars. It can give you an idea about the popularity or the quality of the project.
`NumberOfWatchers`	This function will return the number of watchers. It can give you an idea about the number of people interested in updates to the project.

Table 10. Goal 03 ABS: Ensure every piece of code released is tested thoroughly and confidently.s </> denotes metrics that were successfully automatically implemented.

(a) Metrics generated by GEMS

G	Metrics generated by GEMS
EarlyFault-Detection-Rate	This metric measures the percentage of total detected faults found during the early stages of the testing process. This could be calculated by analyzing the timestamps of fault detection within the test case commit histories and categorizing them into early, middle, and late stages of the testing process.
RegressionTest-Coverage	This metric measures the percentage of code modifications covered by regression tests in each release. This requires tracking data on code modifications (which can be derived from commit histories) and data on the test cases selected for regression testing.
TestSuite-Reduction-Rate	This metric measures the percentage reduction in the number of test cases used to test a release compared to the previous one. This would require tracking the number of tests run for each code release, which can be inferred from the interactions and commit history related to test cases in the repository.
</> Calculate-Feedback-Response-Time	This function calculates the mean time taken to respond to feedback such as bug reports or code review comments. Inputs to the function could be an array of timestamps for when feedback was provided and an array of timestamps for when the feedback was addressed.
Calculate-Pipeline-Efficiency	This function calculates the efficiency of the Continuous Integration/Continuous Deployment (CI/CD) pipeline by computing the mean time taken between each commit to being successfully built and tested in the CI/CD environment. Inputs to the function could be an array of timestamps for each commit and an array of timestamps for when the commit was built and tested.
MeasureTest-Coverage	This function measures the percentage of codebase that is covered by automated tests. Inputs to the function could be the total number of lines in the codebase and the number of lines that have been tested using automated tests.
AutomatedTest-Coverage	This metric will calculate the percentage of the code that is covered by automated tests. It can be computed by dividing the number of lines of code executed during automated testing by the total lines of code in the system.
</> Performance-Issue-Frequency	This metric calculates the frequency of performance-related issues in the software system. It can be computed by dividing the total number of performance-related issues by the total number of issues opened in a given time period.
SelfHealing-Events-Count	This metric calculates the number of instances where the software system has automatically detected and rectified faults. It requires tracking events where self-healing protocols are triggered in the system.

(b) Metrics generated by Vanilla

V	Metrics generated by Vanilla
CodeChurn	Code churn is the amount of code that is rewritten or deleted soon after it's written. High code churn may indicate that the team is struggling with the implementation or there are a lot of bugs in the code.
CodeCoverage	This is a measure of the amount of code that is covered by automated tests. High code coverage is a good sign that the team is testing thoroughly.
CodeQuality-Metrics	Analyze the code for complexity, maintainability, duplications, and issues in the code. This will help to understand the quality of code that is being produced.
Commit-Frequency	This shows the activity and pace of the team. More frequent commits mean the team is actively working and delivering.
IssueMetrics	The number of open vs closed issues, the time taken to close them, and the issue resolution rate can provide insights into the team's efficiency and effectiveness.
PullRequest-Metrics	Analyzing the number of pull requests, their acceptance rate, the time taken to merge them, and the number of comments on them can provide insight into the team's development and review process.

Table 11. Goal 3 CPX: Establish a multi-layered testing framework that incorporates unit testing, integration testing, and automated end-to-end testing, ensuring maximum coverage and efficiency. </> denotes metrics that were successfully automatically implemented.

(a) Metrics generated by GEMS

G	Metrics generated by GEMS
`AutomatedTest-Ratio`	This calculates the ratio of automated tests to total tests. The data required would be the number of automated test cases and total test cases. Both of these can be inferred from commit records and issue data.
</> `Defect-Density`	To compute the defect density, count how many issues have been raised (implying a defect was found) divided by the total size of the codebase (which can be approximated by the number of commits or lines of code).
`TestCoverage-Ratio`	This metric should calculate the ratio of the number of unit tests to the number of functions or methods in the code. The data required would be the number of unit test cases (which can be inferred from the commits mentioning addition or modification of test cases) and the number of functions or methods (which can be inferred from the code within the commits).
`AutomatedEndTo-EndTest-Frequency`	This metric measures how frequently automated end-to-end tests are carried out. It can be calculated by monitoring the testing suite, and recording the timestamps of each automated end-to-end test.
`IntegrationTest-Failures`	This metric captures the number of integration test failures over a given period. This could be calculated by examining the logs of the testing suite, identifying instances of failures in integration tests.
`UnitTest-Coverage`	This metric measures the percentage of code that is covered by unit tests. This can be calculated by identifying the sections of code that are executed during unit tests, and dividing by the total lines of code.
`CalcCI-CD-Integration-Frequency`	This function measures the frequency of integration in the CI/CD pipeline. It can be determined by the commit history and the occurrence of automatic builds and tests. Each cycle in the CI/CD pipeline triggered by a commit can be considered an integration instance.
`CalcTest-Coverage`	This function will calculate the test coverage ratio, defined as the number of lines of code executed during testing divided by the total number of lines in the codebase. This can be inferred by the commit history and the areas of the codebase that changes are being made to. Any change that involves the test suite can be counted towards the lines of code tested.
`CountDocumented-Tests`	This function counts the number of test cases that have corresponding documentation, inferred from the commit history. Any commit that includes changes or additions to the documentation and is linked to a specific test can be counted.

(b) Metrics generated by Vanilla GPT4API

V	Metrics generated by Vanilla
`CodeQuality`	Tools like CodeClimate can provide a maintainability index for the repository which can indicate the quality of code written by the team.
`CommitActivity`	This refers to the frequency of updates to the repository. Regular commits indicate a team's active involvement and consistent work.
`Contributors`	The number of contributors can give an idea of the size of the team and how many people are actively working on the project.
`Issues`	The number of issues opened and closed can give an indication of the team's ability to identify and resolve problems.
`PullRequests`	The number of pull requests created and merged can be an indicator of the team's collaboration level and their ability to accomplish tasks.
`TestCoverage`	If the team has integrated any code coverage tools like Codecov, Coveralls with their Github repository, you can check the percentage of the code base covered by their tests. This would directly indicate how seriously they take testing.

Table 12. Goal 04 ABS: Deliver high-quality products on time, every time, without burning out our team. </> denotes metrics that were successfully automatically implemented.

(a) Metrics generated by GEMS

G	Metrics generated by GEMS
</> `Agile-Adherence`	This function calculates the team's adherence to agile practices by measuring the frequency of changes made (number of commits), the responsiveness to issues (time taken to close an issue after it is opened), and the degree of customer collaboration (number of interactions with users or clients on issues and pull requests).
`Communication-Frequency`	This function calculates the frequency and diversity of communication between team members. It measures this by counting the total number of interactions among team members (comments on commits, issues, and pull requests) and the number of unique individuals involved in these interactions in a given period.
`TeamWellbeing`	This function assesses team well-being by looking at indicators like frequency of late-night commits (as potential signs of overtime or burnout) and the distribution of work among team members (variability in the number of commits made by each team member).
</> `Delivery-Time-Efficiency`	This metric should represent the average time taken for issues or pull requests marked with specific labels (indicating the type and/or complexity of the task) to close. A lower average time may indicate efficient delivery. This metric should control for the complexity as inferred by labels associated with issues or pull requests.
`FeedbackLoop-Efficiency`	This metric should quantify the average response time to issues or pull requests. A shorter average response time would indicate that feedback is provided more quickly, enabling a rapid feedback loop. This metric should consider each interaction as a possible feedback mechanism - comments, reviews, reactions, etc.
</> `TeamCompositionBalance`	This metric should evaluate the balance of skills within a team. This can be indirectly inferred from the different types of contributions (commits, reviews, comments, issues created/resolved) by different users in a repository. A more diverse distribution of contribution types may indicate a balance of skills within the team.
</> `MeasureCode-Refactor-Frequency`	This metric measures the frequency of code refactors by each team member over a given period of time. It is a count of the number of refactoring-related commits made per developer.
`MeasureCommit-Frequency`	This metric measures the frequency of git commits by each team member over a given time period. It is a count of the number of commits made per developer.
</> `Measure-Issue-ResolutionTime`	This metric calculates the average time taken to resolve issues. It involves timestamp values when issues were opened and the moment they were closed. The resolution time of an issue is the difference between these two timestamps. The average resolution time can be computed over all resolved issues.

(b) Metrics generated by Vanilla

V	Metrics generated by Vanilla
`Burnout-Indicator`	To identify signs of team burnout. This could be measured by sudden decreases in commit frequency, increased time to resolve issues, or increased code churn.
`CodeChurn`	To calculate the amount of code that gets modified over time. Lower churn might mean stable and mature code.
`CodeCoverage`	To assess the percentage of code covered by tests. Higher coverage could imply a lower likelihood of bugs and better quality.
`CodeReviews`	To measure the frequency and thoroughness of code reviews. Regular, detailed reviews can improve code quality and reduce bugs.
`Commit-Frequency`	To measure the frequency and consistency of code contribution. Regular commits may indicate a healthy and active team.
`IssuesResolutionTime`	To gauge how long it takes for the team to resolve issues. Shorter times could suggest efficiency and good problem-solving skills.
`OpenToClosed-IssueRatio`	To understand the balance between new issues and resolved issues. A lower ratio may indicate a team that effectively handles issues.
`PullRequest-Acceptance-Rate`	To evaluate the percentage of pull requests that are accepted. A higher rate may reflect a team that produces high-quality code.
`Release-Frequency`	To track how often new versions or updates are released. More frequent releases might suggest a team that can deliver products on time.

Table 13. Goal 04 CPX: Adopt an Agile project management approach, facilitating flexible planning, progressive development, early deployment, and continuous improvement. </> denotes metrics that were successfully automatically implemented.

(a) Metrics generated by GEMS

G	Metrics generated by GEMS
AdaptiveIterationScore	[…] adapting to changes and improvements in their agile methodology. […] by tracking the changes in the number, type, or magnitude of features committed in each iteration (which can be represented by commit records within a certain timeframe). More variation in feature commitments might suggest a more adaptive approach.
CooperativeCommunicationScore	[…] extent of communication and collaboration among all stakeholders. […] summing up the total number of user interactions from commit comments, issue discussions, and pull request reviews, then dividing by number of unique users. […] pulled from the comment, issue, and review tables, respectively. […]
UserRequirementFulfilmentRatio	[…] quantifies how well the development efforts are aligning with user requirements. […] comparing the number of closed issues linked with a user story or a use case against the total number of issues created. Issues can be identified in the issue table with status and link to user story (if available), exact metric calculation may vary based on data available.
CalculateUrgencyIndex	[…] calculate an urgency index by evaluating the frequency and intensity of discussions related to project development and enhancements in project comments, issue discussions, and pull request reviews. […] gauged through sentiment analysis and keyword usage.
DetermineClarityOfVision	[…] measure the clarity of vision within the team by analyzing the consistency of language used in project descriptions, PR descriptions, and issue descriptions for project development with Agile principals. Natural language processing can be applied to provide a metric for this purpose.
MeasureEmpowermentLevel	[…] assess the empowerment level of the team members by examining the distribution of contributions(stats of code contributions, PRs, reviews made etc.) and decision making(with open issue discussions and closed PRs) amongst the team members. […] calculated as a distribution metric for individual contributors.
CalculateIterativeDevelopment	[…] average time taken for each development iteration, which can be inferred by the time difference between consecutive commits by the same user. The average should be calculated across all users over a specified period of time.
MeasureCommunicationEffectiveness	[…] effectiveness of communication in a team by calculating the average number of comments per issue opened within a specific period. […]
TrackContinuousImprovement	[…] the ratio of closed issues to the total number of issues opened within a specified period. A higher ratio indicates that the team is effectively addressing and resolving issues, signifying continuous improvement.

(b) Metrics generated by Vanilla GPT4API

V	Metrics generated by Vanilla
BranchingStrategy	This function analyzes the team's branching strategy. In Agile, developers often work on separate branches and merge their changes frequently, promoting flexible planning and early deployment.
CodeReviewProcess	This function assesses how often code is reviewed and who is involved in the code review process. Regular code reviews are an important part of Agile methodologies, promoting continuous improvement and high-quality code.
CommitFrequency	This function provides information on how often the team commits to the repository. Agile teams often make smaller, more frequent commits, which allows for more flexible planning and continuous improvement.
ContributorDiversity	This function measures the diversity of contributors to the repository. Agile teams often have a wide range of contributors, reflecting their emphasis on teamwork and collaboration.
IssueResolutionTime	This function calculates the average time taken to close issues. Agile teams should be able to respond quickly to issues, reflecting their capacity for flexible planning.
PullRequestVolume	This function measures the number of pull requests made by the team. Frequent pull requests are indicative of early deployment and progressive development.
TestCoverage	This function provides information about the percentage of the codebase covered by automated tests. High test coverage is usually a sign of an Agile team, as this allows for continuous improvement and reduces the risk of regressions.