Industrial Deployment of an AI Multi-Agent System for Requirements-Driven Code Verification

Paul Baker
JPMorganChase
London, UK

Blanca Manu
JPMorganChase
London, UK

Rebecca Moussa
JPMorganChase
London, UK

Federica Sarro
University College London
London, UK

Abstract

Late-stage defect discovery, often rooted in ambiguous requirements, significantly increases remediation costs especially in regulated industries such as fintech.

We present ARC-V, a multi-agent AI system deployed at JPMorganChase that shifts quality assurance upstream by operationalising Large Language Models for automated requirement and code verification. ARC-V utilizes specialised agents to (1) assess requirement tickets against organisational standards in order to provide actionable remedial guidance; (2) verify code against requirements in order to predict defects and offer commit-level feedback; (3) continuously monitor agent performance and adoption.

Post-production deployment results at JPMorganChase show that ARC-V greatly increased the quality score of user story fields with 'value statements' and 'acceptance criteria' achieving score improvements of 8.5 and 4 points, respectively. Crucially, ARC-V achieved a 79% early defect discovery rate, identifying the vast majority of production-escaping bugs before testing.

These results validate a requirements-centric, AI-driven approach to scalable software quality assurance in complex environments.

CCS Concepts

• **Software and its engineering → Software verification and validation**; **Requirements analysis**;

Keywords

Requirements Engineering, Acceptance Criteria, Software Testing

ACM Reference Format:

Paul Baker, Blanca Manu, Rebecca Moussa, and Federica Sarro. 2026. Industrial Deployment of an AI Multi-Agent System for Requirements-Driven Code Verification. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3803437.3805221

1 Introduction

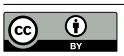
Software quality assurance activities have historically been concentrated late in the software development lifecycle, where defects are expensive to detect and costly to remediate. Techniques such as system testing, integration testing, and contract testing have traditionally served as safety nets for issues that originate much earlier—often in incomplete, ambiguous, or inconsistent requirements. Although Agile and DevOps practices have reduced cycle times, they have not fundamentally changed the economics of late defect discovery, nor addressed the persistent challenge of validating implementation intent against evolving requirements. In fact, defects that escape to production are frequently linked to bad, ambiguous, or misunderstood requirements.

Recent advances in Artificial Intelligence (AI), and Large Language Models (LLMs) in particular, present an opportunity to fundamentally reshape how quality is engineered across the lifecycle[1, 3]. However, automated requirement quality assurance as well as automated requirement-driven code verification remain largely underexplored in industrial contexts [2, 6, 20].

We conjecture that rather than treating requirements as static, human-interpreted documents, LLMs enable requirements to become continuously analysable artifacts—capable of being scored, monitored and validated. This shift enables quality assurance to move upstream, where defects can be identified earlier, corrective action is cheaper, and traceability can be preserved. In this paper, we present ARC-V, a requirement-driven, multi-agent AI system that operationalises LLMs to shift defect discovery earlier in the process and enhance the traceability of compliance on a large scale. ARC-V comprises: (1) ESRA, the Enhancement of Software Requirements Acceptability agent, which evaluates requirement tickets against organizational standards (value statement, assumptions, scope, acceptance criteria) and generates actionable guidance to remedy gaps; (2) AC², the Acceptance Criteria and Code agent, which verifies code against specified requirements, predict defects, and provides targeted commit-level feedback; and (3) PA, the Performance Assessor agents, that continuously measure the adoption of ESRA's recommendations, the developer activity following AC² feedback, and the accuracy of AC²'s defect predictions.

Deployed within enterprise workflows at JPMorganChase, ARC-V is automatically triggered by ticket creation and readiness events, it posts structured feedback directly into Jira tickets, and maintains standardised audit trails.

We carried out a pre-production deployment evaluation to assess feasibility and effectiveness of each of the agents composing ARC-V, as well as a post-production one, where performance is continuously automatically tracked through user-friendly dashboards.



FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805221

Since the results obtained pre-production were positive, we moved ARC-V to the production phase for the Chase mobile development team, encompassing more than 90 software engineers.

The results gathered post-production deployment, show that user story fields such as 'value statement' and 'acceptance criteria' can be greatly improved (8.5 and 4 points, respectively, to date), and business and technical requirements fields also achieve measurable gains. The overall user story quality score rose from ≈4 to ≈6 on average. Notably, ARC-V also achieved an early defect discovery of 79%, meaning that the vast majority of bugs which escaped in production could have been prevented by addressing ARC-V's prediction, with corresponding reductions in rework and verification effort.

Our industrial evaluation shows substantial benefits: ESRA improves Jira ticket clarity and completeness, reducing downstream issues and rework; AC²'s commit-level assessments help developers address requirement–implementation mismatches prior to testing; PAs provide engineers with up-to-date dashboards that quantify in real-time adoption, post-feedback commit behavior, and predictive performance of each agent.

By treating requirements as the primary source of truth and closing the verification loop with measurable outcomes, ARC-V demonstrates a practical path to AI-enabled, requirements-centric software quality assurance in complex, regulated environments.

2 Industrial Context

JPMorganChase is a leading global financial services firm and the largest bank in the United States. Headquartered in New York City, it is a multinational institution that serves millions of consumers, small businesses, and many of the world's most prominent corporate, institutional, and government clients. The company operates under two primary brand names, each targeting a different type of client: Chase (Consumer & Community Banking) which handles personal banking, and J.P. Morgan (Wholesale & Investment Banking), which handles high-level finance and wealthy individuals. Chase focuses on personal banking, lending, credit cards, and it has recently expanded as a digital-only bank in the United Kingdom. Specifically in 2021, JPMorganChase launched Chase, a digital-only retail bank specifically for UK consumers. Unlike its US counterpart, Chase in the UK has no physical high-street branches, and it operates primarily through a mobile app. Since its launch in September 2021, the digital-only bank has grown rapidly, often being cited as one of the fastest-growing fintech apps in British history. As of February 2025, the Chase UK app has over 2.5 million users, and as Chase launches in other European countries, that number is expected to grow quickly. Given that the mobile app is the primary channel for end consumers, it is clear that the quality of the user experience has a significant impact on the success of the bank.

At JPMorganChase, we follow an Agile development process in which requirements are authored as user stories and tracked in the Jira system (i.e., each user story corresponds to a Jira ticket). The ticket allows teams to track the user story through its entire development lifecycle (i.e., To Do, In Progress, Done), assign it to team members, set a priority, track approvals, and link it to other relevant requirement artifacts or documentation in Confluence.

Within JPMorganChase, there are multiple engineering domains: back-end services, mobile, web, data, DevOps, CRM, and AI. All changes made across these domains must adhere to company controls (often driven by regulatory requirements). For example, we have two controls relating to requirements: (1) we must document requirements in Jira, and (2) requirements must be approved prior to implementation.

The process of tracking and approving requirements in Jira is consistent across all domains; however, the artifacts that can be used to supplement requirements in Jira differ between these domains. Table 1 shows the structure of a user story for mobile development which includes the following fields: a value statement, business and technical requirements, and clearly defined acceptance criteria.

The user story format may vary across teams—for example, mobile application teams often differ from back-end services in structure and detail—but all stories undergo the same governance: as part of internal controls, a Product Owner reviews and approves requirements before implementation begins.

Once approved, developers implement software, which is subsequently peer-reviewed (another company control) and then tested. All changes proceed through continuous integration/continuous delivery (CI/CD) pipelines with automated checks, and deployment to production occurs only after every check has passed.

Table 1: Structure of User Story for Mobile development

Field	Required?	Description
Value Statement	*Y*	Anyone picking up the story should understand the "value-add" and what it's trying to achieve. Format: As a.. I want.. So that..
Acceptance Criteria	*Y*	Every story should have at least two Acceptance Criteria to allow for adequate dialogue around testing effort and ensure proper scenario coverage.
Assumptions	*N*	Team may add assumptions if useful.
Out of Scope	*N*	State what's not in scope if useful.
Business Requirements	*Y*	The story should clearly state what is being done functionally as part of the story (i.e. scope). Any updates, refinements to a journey, or new requirements should be included, allowing devs and testers to understand the business objectives. The following should also be included: any accessibility requirements, content requirements, feature flagging, design requirements.
Technical Requirements	*Y*	The story should include enough detail on what needs to be done technically to achieve the business requirements/objectives.

3 ARC-V: AI Multi-Agent Requirements-Driven Code Verification

Figure 1 illustrates our AI multi-agent system, ARC-V, for verifying code against requirements. When requirements are ready for assessment, the first agent, ESRA, evaluates the quality of

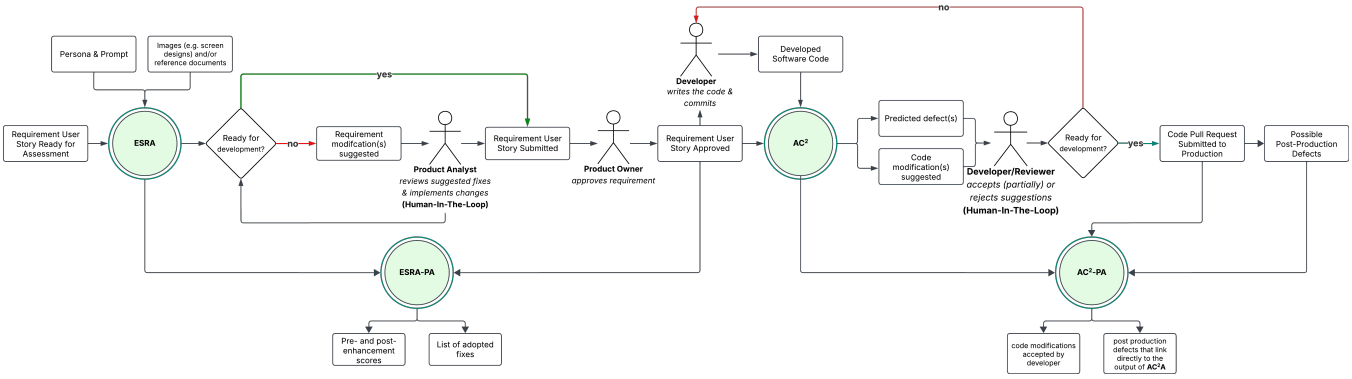


Figure 1: ARC-V Workflow

human-authored requirements and proposes enhancements to both the requirements and their acceptance criteria. The team reviews the proposed changes and updates the Jira ticket accordingly. All changes to the requirements ticket are reviewed by the Product Analyst as part of company controls, acting as an human-in-the-loop (HITL) control for changes suggested by ESRA. After the requirements have been implemented as software, the resulting code and its associated requirements are processed by AC². This agent evaluates whether the developed code correctly implements the requirements and, if it does not, it recommends specific code modifications to achieve compliance with the requirements. The developers review the suggested changes and either reject or accept them. Once all changes have been reviewed, the code undergoes a peer review process, which consists of at least two software engineers thoroughly inspecting the committed code and approving it before it can be merged into the main codebase. This peer review is dictated as part of company controls and acts as a HITL control for code changes suggested by AC². To enable continuous validation and business-impact assessment of ESRA and AC², two Performance Assessor (PA) agents are employed. The ESRA-PA measures the number of ESRA suggested requirement enhancements adopted by product teams and quantifies the uplift in requirement-quality scores. The AC²-PA determines which AC²-recommended code changes were accepted and analyses post-implementation defects related to the requirements to assess whether any of AC²'s suggestions would have prevented those defects from escaping into production.

In the following subsections, we describe each of the four agents.

3.1 The ESRA Agent

The ESRA agent is designed to systematically improve the quality of software requirement tickets by automating feedback on key requirement fields. ESRA operates at the earliest stage of the development cycle, focusing on the clarity, completeness, and consistency of requirement documentation before code implementation begins.

ESRA integrates directly with the project management system (i.e., Jira) and is triggered automatically when a new Jira ticket is created. It retrieves the ticket's content and analyses each field against a predefined organization-specific rubric. These rubrics encode best practices and standards, and are tailored for different teams and domains within the organization. For example, the criteria for front-end mobile (see Table 1) emphasise value statements, business and technical requirements, and acceptance criteria, while the Salesforce team's rubric includes additional fields such as positive/negative scenarios, access specifications, data specifications, and types of testing. Listing 1 shows an excerpt of the prompt used by ESRA for the user story mobile front-end. It is worth noting that due to confidentiality we cannot report all prompts verbatim, but we strive to give a comprehensive description wherever possible.

Listing 1: An excerpt of the ESRA prompt

```
You are a requirements engineer tasked with reviewing and enhancing
      the quality of front-end mobile user stories/requirements. Your
       goal is to suggest concise, value-adding lines that can be
      appended to the existing user story to improve its clarity,
      completeness, and alignment with project goals. Focus on
      enhancing the following areas: Value Statement, Business
      Requirements, Technical Requirements, and Acceptance Criteria.
      Do not alter or remove existing lines; only suggest additions.
      Avoid duplicating existing content.
---
Guidelines for each section:

Value Statement:
- Ensure every story has a clear value statement in the format: "As
      a ... I want.. So that.." [...]

Business Requirements:
- Clearly state the functional scope, including any updates, journey
       refinements, or new requirements.
- Always consider: accessibility requirements, content requirements,
       feature flagging, design requirements. [..]

Technical Requirements:
- Specify technical steps needed to achieve the business objectives.

Acceptance Criteria:
- Every story must have at least 1 acceptance criteria, in the
      format: "Given.. When.. (And..) Then.. " [...]
- Acceptance criteria should enable clear testing and scenario
      coverage. [...]
-----
Input: {jira_ticket}
-----
Output Instructions: [...]
```

The prompt is composed of the role description, including instructions on what to achieve, the guidelines for filling in each field of the user story, the actual Jira ticket, and precise instructions on how to present the output. Moreover, ESRA assigns a quality score

to each ticket through the prompt shown in Listing 2. ESRA judges the quality of a user story based on four required criteria: Value Statement, Business Requirements, Technical Requirements, and Acceptance Criteria. Each criterion is scored on a scale from 0 to 10, with explicit instructions for scoring (e.g., a missing value statement receives a score of 0). While the ESRA agent is prompted to ensure a certain structure mandated by JPMorganChase's definition of ready, its scoring gives more importance to the content of the requirement rather than the structure. ESRA is also able to correctly classify and categorise the aspects of the requirement (i.e., Value Statement, Business Requirements, Technical Requirements and Acceptance Criteria) even if they are not placed in their corresponding Jira fields. The output is a JSON object containing both the overall score and individual scores for each criterion.

Listing 2: Excerpt of ESRAprompt assesing user story quality.

```
You are a requirement engineer who is asked to review the quality of
      front end mobile user stories/requirements.
How good is the quality of the User Story on a scale from 1 to 10
     based on the following required criteria:

- Value Statement: [...] All stories should have a clear Value
     Statement, which should be written in the following format: "As
      a.. I want.. So that..". If it is empty or with a placeholder
     value, give it a score of 0.

- Business Requirements: [...]
- Technical Requirements: [...]
- Acceptance Criteria: [...]
---
User Story Details:
- Summary: {summary}
- Value Statement: {value_statement}
- Description: {description}
- Acceptance Criteria: {acceptance_criteria}
{epic_details}
---
Output Format: Reply only with a JSON in this format, ensuring all
     strings are properly escaped: [..]
```

ESRA's output is posted directly on the ticket, providing stakeholders with actionable suggestions to refine and enhance the ticket's content. For every ticket, ESRA generates a structured feedback comment that highlights potential gaps, ambiguities, or areas for improvement in the requirements. Figure 2 shows an example.

3.2 The AC² Agent

The AC² agent automates the verification of code against specified requirements and acceptance criteria, enhancing reliability and efficiency in the software development lifecycle. AC² is triggered once the implementation for a given ticket is completed and marked as ready for testing. Its primary objective is to ensure the delivered code faithfully meets the documented requirements and acceptance criteria, while proactively identifying potential defects before the code progresses to manual review or production.

The AC²agent's reasoning and workflow are illustrated in Figure 3, which depicts the chain-of-thought from initial input gathering to the generation of actionable feedback. This diagram clarifies how the agent processes both code and requirements, classifies findings by risk and certainty, and formats its output for developers. The agent's workflow involves gathering committed code and ticket information, identifying potential code issues, evaluating acceptance

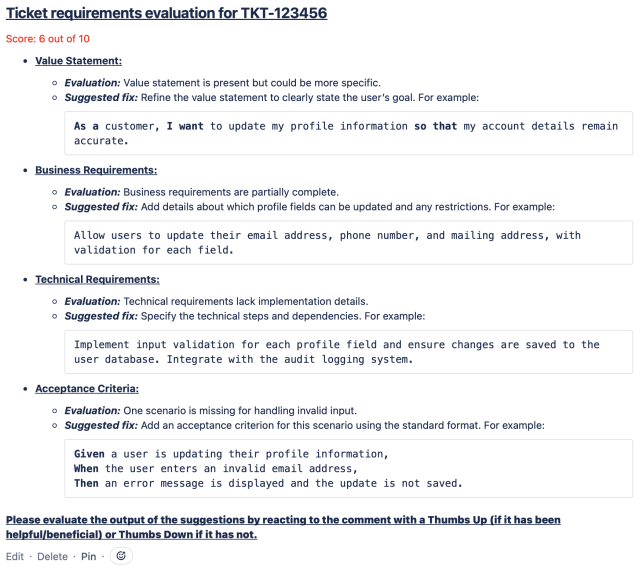


Figure 2: Example of ESRA feedback showing evaluations and actionable suggestions for a user story.

criteria, and checking for missed requirements—including accessibility standards. Each step is supported by a risk and certainty classification, ensuring that only the most relevant and actionable feedback is delivered to developers.

When AC² is triggered, it follows a sequence of four steps, each guided by a dedicated prompt:

(1) *Code Analysis for Defect Prediction*: This prompt asks AC² to examine the code changes associated with the ticket to identify potential issues and provides AC² with all files modified in the relevant commits. For each file, the entire content is provided for context, along with a diff marking added lines with a + and removed lines with a -, following the standard unified diff format. This allows AC² to understand both the local changes and their broader context within the codebase.

(2) *Requirements-driven Code Verification*: This prompt asks AC² to check whether the code fulfills the requirements specified in the Jira ticket. The prompt (see Listing 3) includes the full text of the requirements and acceptance criteria, instructing the agent to map each criterion to the relevant code changes and to flag any mismatches or omissions.

(3) *Standardised Requirements Compliance*: This prompt asks AC² to review the code for compliance with company-wide standards, such as accessibility or security policies. These standards are included in the prompt as additional guidelines, ensuring that organisational policies are consistently enforced.

(4) *Feedback Summarisation*: This prompt asks AC² to compile its findings into a structured feedback report for the developer. Each finding is classified by risk (likelihood of introducing a bug) and confidence level (certainty of assessment), with only high-risk, high-confidence items being included in the feedback.

The final output is formatted as both an HTML file and a Jira comment, streamlining integration into existing development workflows. It is communicated directly to developers, enabling rapid

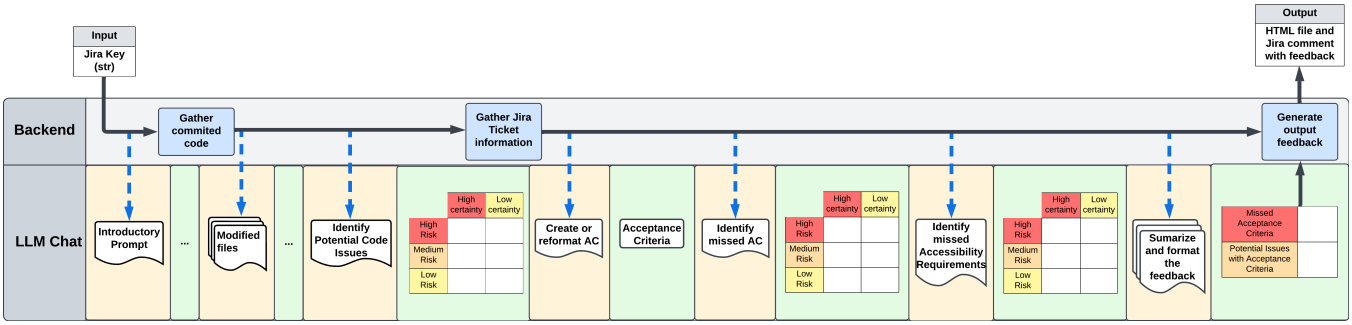


Figure 3: AC² Agent Chain-of-Thought Diagram. The blue arrows depict the points of interaction with the agent.

Potentially Missed Acceptance Criteria:

- ***Ensure password reset notifications are sent to the user***: The code changes do not include logic to notify users when their password is reset. This could result in users being unaware of account changes, which is a security risk.

Potential Code Issues:

- ***Ensure password reset notifications are sent to the user:***
  - *File*: UserAccountService.java
    Lines: 45-60
    Issue: The password reset function updates the user's password but does not trigger a notification event. This omission could lead to missed security alerts for users.
    Suggested Fix: Add a call to the notification service within the password reset function to ensure users receive an email when their password is changed.

Please react with a thumbs up if you found this information useful, or a thumbs down if not. Feel free to reply to this comment with any additional feedback.

Edit · Delete · Pin ·

Figure 4: Example of AC²'s feedback, including identification of missed acceptance criteria and code issue analysis.

iteration and improvement. The output is structured to clearly distinguish between missed acceptance criteria and potential code issues, providing actionable recommendations for each. It provides clear rationale for its assessments to support developer understanding and trust in the feedback process. Figure 4 shows an example of the output. Developers can review the feedback, implement suggested changes, and resubmit the ticket for re-evaluation, creating a feedback loop that supports continuous quality improvement.

Listing 3: Excerpt of AC² prompt for requirements-driven code verification

```
The code modifications were made to address a Jira ticket, which
      outlines its objectives through a list of actionable items or
      criterions. Please evaluate whether the commits satisfy the
      following:
-----
Actionable Items: '{acceptance_criteria}'
-----
To effectively review the code and determine whether each item has
      been met, follow these steps:
1. Review Each Item: [...]
2. Determine the Status of Each Criterion: [...]
3. Indicate Need for Further Context: [...]
4. Specify Confidence Value: [...]
5. Identify Faulty Methods or Parts of the Code: [...]
6. Document Findings: [...]
```

3.3 The PA Agents

To ensure that the ESRA and AC² agents deliver measurable improvements and meet their intended objectives, we introduce two Performance Assessor (PA) agents. Each PA is designed to quantitatively assess the effectiveness of its corresponding agent-ESRA or AC²-by analyzing their outcomes and providing objective metrics. These assessors play a crucial role in continuously validating the impact of automated feedback and verification within the development workflow, offering insights into the level of contribution of the agents to quality assurance and process efficiency.

3.3.1 ESRA-PA. The ESRA-PA, shown on the bottom left of Figure 1, is designed to measure how effectively ESRA achieves its core objective: improving the quality of project tickets. To directly track this goal, the ESRA-PA evaluates the clarity and completeness of ticket's content before and after ESRA's feedback is provided.

A key component of this agent is its ability to analyze the history of changes made to each ticket in order to quantify the adoption of the feedback. The PA works by identifying all tickets that have received feedback from ESRA. For each of these tickets, it examines the ticket's change history to determine whether any modifications were made after the feedback was provided. By comparing the content of the ticket before and after these changes, the PA assesses whether the updates reflect the adoption of ESRA's suggestions - either through directly copying or by incorporating the recommended content into the relevant fields.

The ESRA-PA uses a prompt for *Feedback Adoption Identification* (see Listing 4), which is designed to detect which sentences from the agent's feedback were directly copied into the updated Jira ticket fields. It compares the 'from' and 'to' states of each ticket field after feedback is given, and outputs a structured JSON indicating which sentences were adopted, in which field, and at what time. The prompt strictly instructs the model to include only sentences that are directly copied, and to remove any special characters for consistency. Otherwise, if no feedback is adopted, the output clearly states this. The ESRA-PA agent outputs its decision (namely, whether a developer has adopted ESRA feedback) along with an explanation describing the reasoning behind such a decision. These explanations are essential, as they enable us to understand and audit the agent's thought process as well as help increase engineers' trust in adopting AI-generated suggestions. Table 2 shows examples of tickets that have been processed by ESRA and marked as

adopted by the ESRA-PA agent (i.e., the ESRA-PA agent identified that a developer has refined a given requirement by adopting the suggestions made by ESRA). Each row indicates the development team (*Squad*), the ticket's unique identifier (*Ticket*), which part of the ticket was changed (*Field Modified*), the exact suggestion from ESRA that was incorporated (*Incorporated Suggestion*). The explanation provided by the agent clarifies why it believes the feedback was adopted, helping engineers verify the agent's reasoning. The ESRA-PA agent also provides the engineers with a dashboard summarising the results in terms of average quality scores for all tickets ESRA examined and for all those tickets that have adopted ESRA's suggestions, therefore comparing tickets quality before and after the changes suggested by ESRA are accepted by the developers.

Listing 4: Excerpt of ESRA-PA prompt assessing adoption

```
Purpose: The goal of this task is to identify sentences from the
      feedback that were directly copied into the "to" part of the
      changes made to a Jira ticket, ensuring they are not present in
       the "from" part. This helps in understanding how feedback is
      being utilized in ticket updates.

This is the feedback given for the Jira ticket: {feedback_body}

These are the changes made to the ticket after the feedback was
      given: {changes}

Task: Identify the sentences that were directly copied from the
      feedback and included in the "to" part of the changes, ensuring
       they are not present in the "from" part.

Instructions:

- Under 'feedback_adoption', include strictly only the sentences
      that are directly copied from the feedback and appear in the "
      to" part of the changes, but not in the "from" part. Remove any
       special characters (such as ', ", &, etc.) from these
      sentences. If no sentences were copied, set that field to 'No
      change copied from feedback.'
- For each dictionary in the "changes" array, the "field" key should
       contain the name of the field in the Jira ticket where the
      copied sentences are included in the "to" part, but not in the
      "from" part.
- If no sentences were copied from the feedback, set the "field" key
       to "N/A".
- Ensure proper JSON formatting and escaping of strings.

Reply ONLY with the following JSON: [...]
```

3.3.2 AC²-PA. The AC²-PA, shown on the bottom right of Figure 1, is designed to assess how effectively AC² achieves its primary objective: preventing the introduction of bugs into the codebase by providing targeted feedback on code changes.

The AC²-PA agent is integrated with Jira and code repositories via API. It analyses all tickets that had received feedback from AC² along with their history, code changes, and feedback adoption, and outputs structured results to dashboards and audit logs for team review, as explained in the following.

To quantify AC²'s effectiveness, the PA agent tracks two key metrics that reflect both the agent's influence on developer behaviour and its predictive accuracy in identifying potential defects.

The first metric, *Commit Activity Post-Feedback*, measures the number of unique contributors who made commits to Jira tickets after receiving feedback from AC². To calculate this metric, the PA first identifies all tickets that received feedback and then examines the ticket history to determine whether any new commits were made following the feedback event. For each ticket with subsequent commits, the agent analyses the newly added code in conjunction with the feedback that was provided. It determines whether the changes directly address the issues or suggestions highlighted by AC², thereby assessing the relevance and influence of the agent's feedback on the development process. By tracking both the occurrence and the nature of these post-feedback commits, the metric provides insight into how effectively AC² urges developers to take corrective action and improve code quality in response to its recommendations.

Table 2: Example of tickets adopting ESRA's feedback

Squad	Ticket	Field Modified	Incorporated Suggestion
Alpha Team	ALP-1001	Business Requirements	Business logic should validate user eligibility for premium features.
Beta Squad	BET-2022	Acceptance Criteria	Scenario 2: When a user submits an incomplete form, display an error message.
Gamma Group	GAM-3050	Value Statement	As a user, I want to receive notifications for important account updates.
Delta Crew	DEL-4100	Technical Requirements	System must log all failed login attempts for security auditing.

The second metric, *Defect Prediction Accuracy*, evaluates the agent's ability to predict defects before they reach production. This is accomplished by tracing bugs that are later discovered in the codebase back to the original Jira tickets and analysing whether AC²'s output had identified the relevant issues. By comparing the agent's predictions with actual defect occurrences, the PA assesses the effectiveness of AC² in flagging potential problems early in the development cycle.

Results based on these two metrics are visually summarised in a dashboard that monitors AC²'s performance. The dashboard provides a clear overview of the number of issues which were proactively fixed as a result of AC²'s feedback and the number of bugs which could have been avoided if the agent's suggestions had been fully adopted.

Similar to the ESRA-PA agent, the AC²-PA agent also produces detailed information about its assessment and the reasoning supporting such assessment. In fact, it keeps track of all tickets where the AC² feedback was addressed by developers' code changes, and includes an explanation of how developers have made such code changes in response to specific AC²suggestion(s). An example of the AC²-PA agent output is shown in Table 3, where each row indicates the development team (*Squad*), the ticket unique identifier (*Ticket*), the feedback item that was addressed (*Feedback Addressed*), the files that were modified (*Files Modified*), and an explanation describing how the code changes matched the feedback (*Explanation*). The explanation is crucial for understanding the link between the feedback and the developer's actions.

Moreover, the AC²-PA keeps track of all cases (i.e., tickets) where the AC²agent had predicted a potential defect that was actually later found in production. This data provides clear traceability

between agent feedback, developer actions, and defect outcomes. An example is provided in Table 4, where each row indicates the development team (*Squad*), the original ticket unique identifier (*Ticket*), the bug ticket created after the defect was found (*Bug Introduced*), the type of defect predicted (*Predicted Defect*), the files involved in the bug fix (*Files Modified*), an explanation showing how the agent's feedback related to the defect and its resolution (*Explanation*). The explanation helps reviewers see whether the agent's prediction was accurate and relevant to the actual bug escaped to production.

The AC²-PA employs two prompts to evaluate the effectiveness of the AC² agent, which we describe as follows:

(1) *Commit Activity Post-Feedback:* This prompt analyzes code commits made for a Jira ticket to determine whether they explicitly address the feedback provided by AC². The agent compares the feedback with the code changes, focusing only on modifications that directly and explicitly correlate with the feedback. The output is a structured JSON object that includes whether feedback was addressed, details of the addressed feedback, file names, commit IDs, timestamps, and concise explanations linking feedback to code changes. The prompt enforces strict criteria to avoid assumptions and only considers changes with high confidence in their correlation to the feedback.

(2) *Defect Prediction:* This prompt assesses whether a bug could have been prevented if specific feedback had been addressed. It reviews the code changes made to resolve a bug, focusing on lines introduced by a particular Jira ticket, and evaluates whether the feedback for that ticket warned about those lines. The output is a structured JSON indicating whether bug prevention was possible, whether feedback warnings were present, and details relating feedback to the cause of the bug. The prompt is designed to ensure explanations clearly link feedback to the bug resolution, and only considers feedback that directly relates to the code.

3.4 LLM Agents' Prompt Design and Settings

All ARC-V agents are implemented in Python, integrated with Jira and code repositories via API, and use the `GPT-4o-mini` model, accessed through the JPMorganChase LLM Suite platform.[1]

Prior to deployment, prompts for all agents were tuned using historical Jira ticket data and known defect cases. This allowed the team to calibrate instructions and output formats based on real-world scenarios and to ensure initial relevance and accuracy. Following deployment, prompts were further refined through iterative pilot deployments and ongoing stakeholder feedback. This process involved continuous adjustment of instructions and output formats to enhance clarity, actionability, and alignment with user needs.

Prompts for the PA agents were iteratively refined by reviewing feedback adoption and scoring results. Adjustments were ongoing until agent outputs closely matched manual human judgment, improving both accuracy and reliability.

To ensure consistency and reproducibility, all agents operated with the temperature parameter set to 0, ensuring identical inputs yield identical outputs. All agent outputs are formatted as structured JSON. Any suggested content is formatted to reflect JPMorganChase's internal requirements and standards. This ensures that agent feedback is consistent, relevant, and immediately applicable across different engineering teams and domains.

To maximise reliability, prompts include explicit instructions to avoid duplicating existing content, to suggest only concise and value-adding recommendations, and to reply strictly in the required format. These constraints, combined with deterministic model settings, significantly reduce the risk of hallucinations and irrelevant output.

The design and configuration of prompts, as well as the selection of underlying language models, are subject to ongoing review and enhancement. As new model versions and best practices emerge, ARC-V agents are updated to maintain optimal performance and alignment with evolving organisational needs.

4 Industrial Evaluation

To conduct the evaluation, we chose the mobile app domain. The rationale was that testing mobile applications and fixing identified defects is generally more expensive than in other engineering domains. The approach taken was to develop and fine-tune prompts using a selected set of requirement tickets. This process was manual and required direct feedback from the teams. Following this initial development, we conducted proof-of-concepts with selected teams-initially soliciting their feedback, and later automating this process using performance assessor agents.

4.1 Pre-production Deployment

Before deploying the ESRA and AC² agents in the JPMorganChase production environment, we conducted a retrospective evaluation using historical Jira ticket data. A total of 84 tickets were examined, representing work from 13 different development teams over the previous year. The sample was designed to reflect typical development activity and team diversity. For ESRA, 35 user stories (i.e., Jira tickets) were randomly selected to ensure a broad representation of requirement documentation practices. For AC², the selection focused specifically on user stories associated with known defects, as a result, 51 tickets were selected by tracing resolved bugs back to their source: a custom tool was developed to analyse the lines of code which were changed to fix a bug, then to find the commits that introduced those lines, and finally to link those commits to the corresponding Jira tickets. To simulate agent deployment, both ESRA and AC²were run on the selected tickets.

The feedback generated by ESRA has been manually reviewed by two groups: the team that developed the agent and the teams that would ultimately use it. The reviewers assessed the feedback based on clarity, usefulness and effort reduction, providing qualitative comments on its value and relevance to their workflow. This was achieved by verifying that ESRA's output matches the changes made on the requirement's description after ESRA's suggested changes. In other words, this tells us that the suggestions are deemed accurate by the software engineers.

For AC², the evaluation was quantitative. The AC² agent was run on tickets associated with previously resolved bugs, and its predictions were compared with the actual bug descriptions and the

[1]LLM Suite is JPMorganChase's proprietary generative AI platform, released in summer 2024 to eligible employees across the firm, providing secure access to advanced language models.

Table 3: Example of tickets adopting AC²'s feedback

Squad	Ticket	Feedback Addressed	Files Modified	Explanation
Alpha Team	ALP-1001	Update Data Validation	DataValidator.java	Added checks for null values in user input as recommended by the agent.
Beta Squad	BET-2022	Improve Error Logging	ErrorLogger.py	Enhanced error logging for failed operations per agent feedback.
Gamma Group	GAM-3050	Refactor API Endpoint	ApiHandler.js	Refactored API endpoint to handle edge cases as suggested.
Delta Crew	DEL-4100	Optimize UI Rendering	Renderer.tsx	Optimized UI rendering logic to address performance issues flagged.

Table 4: Example of defects predicted by AC² for given tickets

Squad	Ticket	Bug Introduced	Predicted Defect	Files Modified	Explanation
Alpha Team	ALP-1001	ALP-1101	Missing Null Check	DataValidator.java	The agent flagged missing null checks, which led to a bug when null data was processed.
Beta Squad	BET-2022	BET-2033	Inadequate Error Logging	ErrorLogger.py	The agent predicted insufficient error logging, resulting in undetected failures.
Gamma Group	GAM-3050	GAM-3060	Unhandled API Edge Case	ApiHandler.js	The agent identified unhandled edge cases in the API, which caused a bug.
Delta Crew	DEL-4100	DEL-4110	UI Performance Issue	Renderer.tsx	The agent predicted performance issues in UI rendering, which were later confirmed as bugs.

code changes made to resolve them. The scoring system measured how many of these bugs could have been avoided or fixed more efficiently if AC²'s feedback had been available at the time. The decision to deploy the agent was based on the results of these evaluations. The deployment was approved once the AC² agent demonstrated that it could have predicted or prevented over 75% of the bugs escaped to production based on the retrospective analysis.

To evaluate the PA agents, we conducted a manual analysis of their output. Specifically, we assessed the output of their performance assessment analysis (e.g., whether ESRA feedback was adopted or a defect was predicted) as well as the explanations they provide for each of the output. By manually reviewing both the results and the explanations (see Tables 2, 3 and 4), one can determine not only if the PA agents are making correct decisions, but also if their reasoning is clear and trustworthy. This dual evaluation is essential for building confidence in the agents' performance. For this evaluation, we randomly selected 20 tickets for each agent, and checked whether the explanations provided by the PA agents were correct and justified the results. This process allows us to directly measure not only the accuracy of the agents' decisions, but also the clarity and reliability of their reasoning. Our findings show that in 80% of the sampled cases, the explanations given by the PA agents were clear, accurate, and aligned with manual assessment. This demonstrates that the agents are not only making correct decisions, but are also able to justify those decisions in a way that is transparent and trustworthy.

4.2 Post-Production Deployment

Based upon the promising results from the pre-deployment evaluation phase, we have deployed ARC-V to a limited cohort of development teams within the mobile domain.

4.2.1 ESRA. To assess the impact of ESRA post-deployment into production, we have continuously monitored ticket quality using the ESRA-PA agent. As explained in Section 3.1, the PA agent automatically tracks all tickets that received ESRA feedback, analyses subsequent modifications, and scores the quality of each user story's

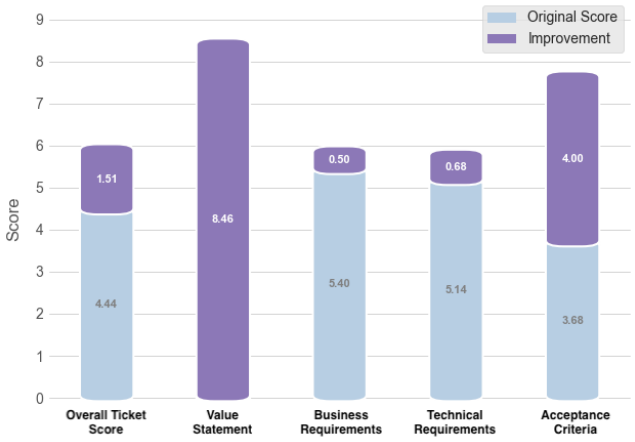
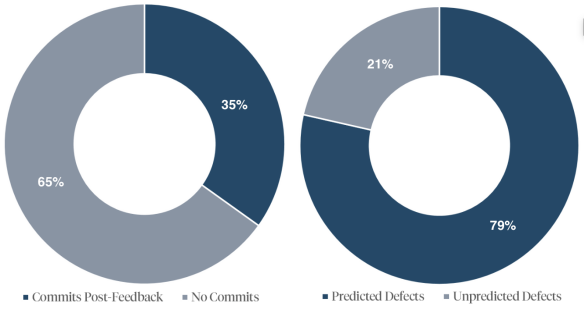


Figure 5: Average Score of user story *before* and *after* ESRA's suggested modifications are adopted by developers. In blue the score of the original human-written user story, in purple the improvement due to the adoption of ESRA's feedback.

field (namely value statement, business requirements, technical requirements, and acceptance criteria) using a standardised rubric. For tickets where ESRA's suggestions were adopted, the PA compared the quality scores before and after the modifications, ensuring unbiased evaluation by scoring both versions independently.

Figure 5 shows the results in form of a stacked bar chart illustrating the average scores for each field *before* and *after* the developer's adoption of ESRA's feedback. We can observe that, notably, the value statement and acceptance criteria fields saw the most significant improvements, with an increase of 8.5 and 4 points, respectively. The business and technical requirement fields also showed measurable gains. The overall ticket score improved by 1.51 points, rising from 4.44 to 6. This shows that ESRA's feedback, when adopted, leads to substantial improvements in the clarity and completeness of ticket documentation.



(a) Adoption Rate **(b) Defect Prediction**

Figure 6: AC²'s Results

These results confirm that ESRA is effective in guiding teams toward higher-quality requirements, particularly in areas that are often overlooked or under-specified. By providing actionable, targeted feedback and supporting consistent standards across teams, ESRA contributes to improved requirements engineering and reduces the risk of downstream defects and rework.

4.2.2 AC². To evaluate the effectiveness of AC² after deployment in production, we have continuously monitored its impact on code quality and defect prevention using the AC²-PA agent. This agent tracks two key metrics: the Commit Activity Post-Feedback and the Defect Prediction Accuracy as described in Section 3.2.

Figure 6a shows the Activity Post-Feedback, in other words, the developers' adoption of AC²'s feedback. We can observe that 65% of the tickets processed by AC² have an associated commit that directly addresses the feedback provided by AC². This is a very positive start, as these early results did not take into account AI agent automation in which suggested code changes are made directly to code repositories, thereby avoiding the need for developers to make code changes themselves. This is significant because developers were not forced to review potential improvements as part of these results by either accepting or rejecting suggested code modifications. We believe this will further increase adoption.

Figure 6b shows the defect prediction results, in other words, the ability of AC² to identify defects at commit time. The higher this ability, the fewer bugs escape to production. We observe that 79% of the tickets predicted as defective by AC² at commit time were found to be in fact defective in production, and therefore, such defects could have been avoided if developers would have had AC²'s feedback available at commit time. This is a significant insight: from our analysis of production defects, approximately 30% of defects that escape into production are related to missed requirements. Therefore, we believe that by using AC², there is a great potential in reducing the overall number of production defects by up to 23%.

5 Impact, Lessons Learned & Future Work

ESRA has shown the capacity to deliver benefits throughout the software development lifecycle. By intervening at the requirements stage, it helps ensure that tickets are well defined and comprehensive, which in turn prevents downstream issues and improves overall quality. By automating the feedback process, ESRA shifts all tickets towards a consistent standard and style. ESRA enforces uniformity in ticket structure and content, aligning submissions with organisational standards and reducing variability across projects. Automated feedback also accelerates the requirement approval review process, which is mandatory before any requirement is approved for development, and has been shown to improve requirement quality scores.

AC² has proven to be a significant advancement in how we employ quality assurance practices during software development and is a key contribution of this work. We have demonstrated industrial impact in the mobile domain, and next step is to expand and evaluate effectiveness across the other engineering domains (back-end, SalesForce, etc.) at Chase.

Initially, validating the effectiveness of agent-generated feedback was challenging due to limited developer responses. To overcome this, two specialised PA agents were introduced to automatically capture developers (re-)actions following the delivery of feedback. These agents systematically track developer updates to requirements and code, identifying instances where agent suggestions are adopted—such as modifications to tickets or code changes that align with the provided recommendations. This automated approach enables independent and objective measurement of agent impact, reducing reliance on subjective human feedback. The resulting data offers richer and more reliable insights into the adoption and influence of agent suggestions, informing ongoing improvements to ARC-V and guiding future development. This methodology underscores the broader value of automating validation and impact assessment in AI-driven software engineering.

While the system's automation and impact is clear, our experience has highlighted that more work is needed to further increase adoption rates. This is not due to the quality of the agent-generated feedback—which has proven effective where adopted—but rather appears to be a challenge of communication and visibility. For example, feedback comments often receive few reactions, and some developers have reported that they do not always notice the agent's suggestions. These findings suggest that improving the delivery and integration of feedback could further enhance adoption and impact. To address this, we are currently focusing on further refining the agents into an "implementation" version that will make it easier for developers to act on suggestions. For ESRA, this means automatically adding agent-generated suggestions directly into tickets, clearly marking them as agent input and formatting them for easy adaptation or removal. For AC², we plan to leverage GitHub pull request comments to provide actionable code change suggestions that developers can accept and commit with minimal effort.

Finally, all ARC-V agents use the GPT-4o-mini model accessed through the JPMorganChase LLM Suite platform, in future we plan to investigate the impact of the choice of the LLM model.

6 Related Work

In this section we give an overview of previous work investigating the use of LLMs for creating, enhancing or verifying requirements. We refer the reader to existing literature reviews on the use of LLMs for Requirement Engineering [2, 14, 21], LLMs for Software Engineering [3, 7], and Requirements-driven Automated Software Testing [20].

Previous work has explored the *use of LLMs to automatically write requirements, user stories or acceptance criteria* [9, 11, 16, 18, 19].

Krishna et al. [9] carried out a preliminary empirical evaluation of early LLMs (GPT-4 and CodeLlama) to create software requirement specifications for a university club management system, showing that GPT-4 was able to do so, while CodeLlama's results were not as encouraging. Rahman et al. [16] explored the use of GPT-4-turbo to automatically create user stories from software requirement documents. The generated user stories were rated as "Good" and accepted by the majority of the SE experts who participated in the survey, however, they pointed out that there is room of improvement, especially in the Specificity and Technical Aspects categories. Subsequently, Sami et al. [18] explored the use of role-based agents powered by GPT-3.5 and GPT-4o to automate the generation, quality assessment, and prioritization of agile user stories on a single web project. Their results demonstrate that these LLMs can identify core features and streamline requirements analysis, with GPT-3.5 showing particularly high performance. Similarly, Li et al. [11] have proposed a framework leveraging LLMs to simulate agile collaboration and generate AC, showing promising results on a synthetic benchmark. More recently, Wang et al. [19] have reported promising results investigating the use of Retrieval-Augmented Generation to generate AC based on multi-modal requirements data, including both textual documentation and visual user interface information, for an education-focused software system.

Another research line has focused on investigating the *use of LLMs to automatically verify requirements*.

Several attempts were carried out in this direction between 2023-2024, following the release of the first ChatGPT [1, 4, 12, 13]. For example Fantechi et al. [5] conducted an early experiment to test the ability of ChatGPT (GPT-3.5) in finding inconsistency in requirements by comparing its predictions with those obtained from expert judgments by students on a few example requirements documents. While Lubos et al. [12] assessed the ability of LLM to evaluate the quality characteristics of software requirements according to the ISO 29148 standard. Despite these work being preliminary, assessed mainly using synthetic data, or small single benchmark project, and not always involving practitioners, their results were promising and motivated subsequent more mature research.

A more recent line of study has explored the *importance of requirement/task description quality to fully automate code generation.* Larbi et al. [10] presented the first comphrensive empirical study examining the robustness of state-of-the-art code generation models when faced with such unclear requirement descriptions. They found that even minor imperfections in task description can cause significant performance degradation, with contradictory task descriptions resulting in numerous logical errors. Moreover, while larger models tend to be more resilient than smaller variants, they are not immune to the challenges posed by unclear requirements.

Recent proposals *investigate the use of LLMs to automatically identify ambiguous requirement descriptions and fix such ambiguity* [8, 15], with the ultimate goal to automatically generate high-quality code. Jia et al. [8] have proposed to perform automated repair of ambiguous requirement descriptions by reducing code generation uncertainty and better aligning descriptions with input-output code examples. Mu et al. [15] have proposed ClarifyGPT, aiming at enhancing code generation by empowering LLMs with the ability to identify ambiguous requirements and ask targeted clarifying questions. After receiving question responses, ClarifyGPT refines the ambiguous requirement and inputs it into the same LLM to generate a final code solution. These previous work mainly rely on the use of public code generation benchmarks (e.g., HumanEval, MBPP), which might not capture the complexity of industrial projects (see e.g., [17]).

Our work significantly differs from previous studies by proposing an industrial end-to-end AI multi-agent system capable to: (1) automatically validate the quality of user stories with respect to well-defined firm-wide standards; (2) automatically verify that code captures the corresponding requirements and AC in an industrial, real-world fintech setting; (3) suggest code modifications to make the implementation compliant with the requirements and ACs; and (4) periodically measure agents' performance and automatically present these results to engineers in the form of user-friendly dashboards. In doing so, we have introduced continuous, production-grade monitoring of agent performance, a capability largely absent from existing research. Moreover, our work empirically demonstrates that requirement conformance can be linked to production defect rates, rather than treating requirements as a purely upstream concern. Further, we use prompts as tunable, production-grade artifacts whose evolution is driven by empirical feedback. Lastly, we validate our approach in a live fintech environment, addressing scale, controls compliance, and real-world constraints which is absent from prior studies.

7 Conclusions

In this work, we presented ARC-V, an industrially deployed, multi-agent AI system for requirements-driven code verification at JPMorganChase.

ARC-V operationalises LLMs to automate the assessment and improvement of requirements, verify code against acceptance criteria, and continuously measure the impact of these interventions through dedicated performance assessor agents.

Our post-production deployment results demonstrate that ARC-V significantly improves the quality and clarity of requirements documentation, with measurable uplifts in key fields such as value statements and acceptance criteria. Most notably, ARC-V enables early defect discovery, identifying 79% of bugs before they reach production, thereby reducing costly rework and verification effort.

By shifting quality assurance upstream and treating requirements as the primary source of truth, ARC-V offers a scalable, auditable, and requirements-centric approach to software quality in complex regulated environments.

ARC-V's integration into enterprise workflows and its continuous feedback loop empower engineering teams to proactively address gaps and misalignments, fostering a culture of quality and compliance.

Overall, ARC-V confirms the promise of AI multi-agent systems in transforming software engineering practice. By continuing to refine both the content and delivery of agent feedback, we aim to further increase adoption, streamline quality assurance, and enhance the reliability and efficiency of enterprise software delivery.

Acknowledgments

We would like to thank Paul Clark for his enthusiastic support to this work when it was just a vision, Raphael Cohen for his continuous support and Ish Mishra and Martynas Jagutis for their technical contribution to turn this vision into a fully deployed product at JPMorganChase.

References

[1] Chetan Arora, John Grundy, and Mohamed Abdelrazek. 2024. Advancing Requirements Engineering Through Generative AI: Assessing the Role of LLMs. In *Requirements Engineering: Trends and Opportunities with Generative AI*. Springer Nature Switzerland, Cham, 129–148. doi:10.1007/978-3-031-55642-5_6

[2] Haowei Cheng, Jati H. Husen, Yijun Lu, Teeradaj Racharak, Nobukazu Yoshioka, Naoyasu Ubayashi, and Hironori Washizaki. 2025. Generative AI for Requirements Engineering: A Systematic Literature Review. *Software: Practice and Experience* 56, 2 (Nov. 2025), 141–170. doi:10.1002/spe.70029

[3] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[4] Alessandro Fantechi, Stefania Gnesi, Lucia Passaro, and Laura Semini. 2023. Inconsistency Detection in Natural Language Requirements using ChatGPT: a Preliminary Evaluation. In *2023 IEEE 31st International Requirements Engineering Conference (RE)*. 335–340. doi:10.1109/RE57278.2023.00045

[5] Alessandro Fantechi, Stefania Gnesi, Lucia Passaro, and Laura Semini. 2023. Inconsistency Detection in Natural Language Requirements using ChatGPT: a Preliminary Evaluation. 335–340. doi:10.1109/RE57278.2023.00045

[6] Xavier Franch, Cristina Palomares, Carme Quer, Panagiota Chatzipetrou, and Tony Gorschek. 2023. The state-of-practice in requirements specification: an extended interview study at 12 companies. *Requir. Eng.* 28, 3 (April 2023), 377–409. doi:10.1007/s00766-023-00399-7

[7] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 220 (Dec. 2024), 79 pages. doi:10.1145/3695988

[8] Haoxiang Jia, Robbie Morris, He Ye, Federica Sarro, and Sergey Mechtaev. 2025. Automated Repair of Ambiguous Problem Descriptions for LLM-Based Code Generation. In *2025 IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://arxiv.org/abs/2505.07270

[9] Madhava Krishna, Bhagesh Gaur, Arsh Verma, and Pankaj Jalote. 2024. Using LLMs in Software Requirements Specifications: An Empirical Evaluation. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. 475–483. doi:10.1109/RE59067.2024.00056

[10] Maya Larbi, Amal Akli, Mike Papadakis, Rihab Bouyousfi, Maxime Cordy, Federica Sarro, and Yves Le Traon. 2026. When Prompts Go Wrong: Evaluating Code Model Robustness to Ambiguous, Contradictory, and Incomplete Task Descriptions. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. https://arxiv.org/abs/2507.20439

[11] Yishu Li, Jacky Keung, Zhen Yang, Xiaoxue Ma, Jingyu Zhang, and Shuo Liu. 2024. SimAC: simulating agile collaboration to generate acceptance criteria in user story elaboration. *Automated Software Engg.* 31, 2 (June 2024), 56 pages. doi:10.1007/s10515-024-00448-7

[12] Sebastian Lubos, Alexander Felfernig, Thi Ngoc Trang Tran, Damian Garber, Merfat El Mansi, Seda Polat Erdeniz, and Viet-Man Le. 2024. Leveraging LLMs for the Quality Assurance of Software Requirements . In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE Computer Society, Los Alamitos, CA, USA, 389–397. doi:10.1109/RE59067.2024.00046

[13] Dipeeka Luitel, Shabnam Hassani, and Mehrdad Sabetzadeh. 2023. Using Language Models for Enhancing the Completeness of Natural-Language Requirements. In *Requirements Engineering: Foundation for Software Quality*, Alessio Ferrari and Birgit Penzenstadler (Eds.). Springer Nature Switzerland, Cham, 87–104.

[14] Nuno Marques, Ricardo R. Silva, and Jorge Bernardino. 2024. Using ChatGPT in Software Requirements Engineering: A Comprehensive Review. *Future Internet* 16, 6 (2024). https://www.mdpi.com/1999-5903/16/6/180 [Online; accessed January 2026].

[15] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proc. ACM Softw. Eng.* 1, FSE, Article 103 (July 2024), 23 pages. doi:10.1145/3660810

[16] Tajmilur Rahman, Yuecai Zhu, Lamyea Maha, Chanchal Roy, Banani Roy, and Kevin Schneider. 2024. Take Loads Off Your Developers: Automated User Story Generation using Large Language Model. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 791–801. doi:10.1109/ICSME58944.2024.00082

[17] Pat Rondon, Renyao Wei, José Cambronero, Jürgen Cito, Aaron Sun, Siddhant Sanyam, Michele Tufano, and Satish Chandra. 2025. Evaluating Agent-based Program Repair at Google. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. https://arxiv.org/abs/2501.07531

[18] Malik Abdul Sami, Muhammad Waseem, Zheying Zhang, Zeeshan Rasheed, Kari Systä, and Pekka Abrahamsson. 2025. Early Results of an AI Multiagent System for Requirements Elicitation and Analysis. In *Product-Focused Software Process Improvement*, Dietmar Pfahl, Javier Gonzalez Huerta, Jil Klünder, and Hina Anwar (Eds.). Springer Nature Switzerland, Cham, 307–316.

[19] Fanyu Wang, Chetan Arora, Yonghui Liu, Kaicheng Huang, Chakkrit Tantithamthavorn, Aldeida Aleti, Dishan Sambathkumar, and David Lo. 2025. Multi-Modal Requirements Data-based Acceptance Criteria Generation using LLMs. In *ACM/IEEE Conference on Automated Software Engineering - Industry Track*. https://arxiv.org/abs/2508.06888

[20] Fanyu Wang, Chetan Arora, Chakkrit Tantithamthavorn, Kaicheng Huang, and Aldeida Aleti. 2025. Requirements-Driven Automated Software Testing: A Systematic Review. *ACM Trans. Softw. Eng. Methodol.* (Sept. 2025). doi:10.1145/3767739

[21] Mohammad Amin Zadenoori, Jacek Dąbrowski, Waad Alhoshan, Liping Zhao, and Alessio Ferrari. 2025. Large Language Models (LLMs) for Requirements Engineering (RE): A Systematic Literature Review. https://arxiv.org/abs/2509.11446