**Journal of Applied Informatics and Computing (JAIC)**
Vol.10, No.3, June 2026, pp. 2556~2567
e-ISSN: 2548-6861


# Comparative Evaluation of Agentic Workflow Capabilities in AI IDE Agents for Web-Based Learning Media Development

**Erlangga Aditia[1]*, Ulva Elviani [2]***
* Information Systems and Technology Education, Universitas Pendidikan Indonesia
erladitia14@upi.edu [1], ulva@upi.edu [2]

**Article Info**

***Article history:***

Received 2026-04-23
Revised 2026-05-18
Accepted 2026-05-25

***Keyword:***

*Agentic Workflow,*
*AI IDE Agents,*
*Comparative Evaluation,*
*Web-Based Learning Media*

**ABSTRACT**

The rapid development of agentic IDEs calls for evaluation approaches that assess not only final outputs but also the agentic workflow enacted during software development. This study comparatively evaluates the workflow capabilities of four AI IDE agents, namely Cursor, Windsurf, Trae, and Antigravity, within the five-stage benchmark of developing a web-based learning media application, Next-Gen SPLDV. A descriptive comparative evaluation was conducted using five agentic maturity metrics: task decomposition (DC), tool-use effectiveness (TSR), autonomous recovery capability (ARC), human intervention cost (HIC), and time completion efficiency (TCT), complemented by systematic log coding using four workflow markers—planning, execution, verification, and recovery/intervention—triangulated with internal artifacts such as plan files, implementation documents, code snapshots, and walkthrough notes. Within the observed benchmark context, the four systems exhibited distinct workflow trade-off profiles rather than a single dominant pattern. Cursor showed a selective tool-use orientation with compact planning artifacts; Windsurf adopted an exploratory trial-and-repair style with higher tool-use volume and recovery effort; Trae produced detailed code-evolution traces and was efficient in several stages but more vulnerable during database integration; and Antigravity displayed a more structured plan-execute-verify orientation with comparatively lower recovery and intervention demands in this benchmark. The plan-execute-verify and error-response differences were traced to concrete log evidence and recurring failure patterns, including the Prisma v7 breaking change and shell-syntax mismatches, rather than to narrative impression alone. Overall, the evaluation of AI IDE agents is better interpreted as a contextual map of workflow trade-offs rather than the identification of a single winner across all settings.

This is an open access article under the CC–BY-SA license.

## I. INTRODUCTION

The integration of artificial intelligence into software engineering has reshaped how developers write and manage code [1], [2], [3]. Early generations of this technology primarily focused on code completion, namely the reactive provision of syntax suggestions based on the immediate coding context [4]. More recent developments indicate a shift toward a more autonomous paradigm, commonly referred to as agentic programming, in which systems translate high-level goals into a sequence of technical actions with greater independence [5], [6]. Within this paradigm, agents engage in iterative cycles of planning, tool execution, and result evaluation without requiring human intervention at every step [7], [8], [9]. This iterative process is commonly described as an agentic workflow [5].

Recent studies on agentic IDEs and AI coding agents can be grouped into three complementary clusters, yet none fully addresses the need for direct operational evaluation of workflow behavior across comparable AI IDE environments. The first cluster focuses on feature mapping and platform characteristics. For example, Shrivastava [10] provides a baseline comparison of agent components across systems, but the evidence remains largely derived from technical

documentation and therefore does not capture workflow behavior during actual task execution.

The second cluster focuses on standardized code and software-engineering benchmarks. HumanEval evaluates functional correctness in program synthesis from docstrings through unit-test-based assessment [11], while SWE-Bench evaluates issue-resolution capability using real GitHub issues and corresponding pull requests, with generated patches assessed through repository-level tests [12]. These benchmarks are important for assessing code correctness and issue resolution, but they do not primarily examine the end-to-end workflow process of AI IDE agents across staged development tasks.

The third cluster, which is closest to the present study, emphasizes the urgency of evaluating agentic workflow processes and critiques the limitations of current benchmarks, which often remain outdated or rely on single-shot tasks [5], [13]. However, this line of work has not yet provided a comparative instrument that directly evaluates multi-stage workflows under comparable test conditions across systems. As a result, the operational capabilities of AI IDE agents in completing software development workflows autonomously and comprehensively have not been evaluated in a balanced manner. This gap motivates the need for direct evaluation using equivalent scenarios and stages across systems.

To address this gap, the present study develops a descriptive comparative evaluation that directly tests AI IDE agents through a web-based learning media development scenario. This scenario was chosen for two main reasons. First, web-based learning media development requires the integration of application interfaces, processing services, and databases within an interconnected architecture [14], [15], [16]. Such complexity is well suited to testing how agents orchestrate multiple technical components within a standardized workflow across AI IDE agents. Second, as noted by Lyons et al. [17], the development of web-based learning tools is inherently iterative. This condition provides an appropriate testing environment for examining an agent's autonomous recovery capability in a multi-step workflow.

More specifically, this study compares the agentic workflow capabilities of four AI IDE agents: Cursor, Windsurf, Trae, and Antigravity. These systems were selected to represent a range of contemporary software-agent architectures [10]. Cursor represents an AI-first IDE approach with integrated agents, supported by background agents for exploration, planning, and codebase modification with relative autonomy [10]. Windsurf represents the agentic IDE paradigm through its Cascade agent, particularly Cascade Write Mode, which enables multi-file creation, script execution, testing, and debugging within a centralized workflow [10]. Trae is included as an AI IDE agent that positions its SOLO Builder agent as capable of building web applications end-to-end from natural-language requirements [18]. Antigravity represents an agent-first development platform that provides a planning mode for complex task planning through implementation plans, task groups, and artifacts [19]. Including the newer-generation systems, Antigravity and Trae, broadens the comparative scope to rapidly evolving tools while also responding to the literature's concern that evaluation research should not lag behind the pace of commercial tool development [5].

To evaluate these four AI IDE agents in a measurable way, this study defines agentic maturity as the extent to which an AI IDE agent can complete a multi-step software development workflow autonomously, accurately, and consistently. Wang et al. [5] and Bandi et al. [13] converge in identifying tool-use efficiency, recovery from failure, and responsiveness to human feedback as key dimensions in evaluating agent systems. Beyond these three aspects, task planning is identified as an additional relevant dimension because the success of multi-step workflows depends strongly on an agent's ability to decompose an initial goal into structured execution steps [20], [21]. In the literature, this planning dimension is often discussed in terms of planning pattern and dynamic task decomposition [22]. Based on this synthesis, the construct of agentic maturity in this study is operationalized into four core dimensions: (1) task decomposition quality, (2) tool-use effectiveness, (3) autonomous recovery capability, and (4) minimization of human intervention. Time completion efficiency is used as an additional performance indicator.

Based on these operational parameters, this study formulates two research questions. First, RQ1 asks: How do AI IDE agents compare in terms of agentic maturity across task decomposition, tool use, error recovery, human intervention, and time efficiency within a staged development workflow? Second, RQ2 asks: What patterns of agent behavior underlie the observed performance differences among AI IDE agents, particularly in error-recovery strategies and tool-use behavior?

The contribution of this study is framed at three levels to distinguish it from increasingly common evaluations of AI coding agents. First, at the empirical level, it provides a controlled descriptive comparison of four contemporary AI IDE agents--Cursor, Windsurf, Trae, and Antigravity--within the same educational web-application benchmark, stages, prompts, resources, and evaluation criteria. Rather than comparing only product features or final outputs, the study observes how each agent performs across a staged development workflow.

Second, at the methodological level, it operationalizes agentic workflow capability through process-oriented metrics (DC, TSR, ARC, HIC, and TCT) supported by interaction-log evidence and internal artifacts, thereby complementing prior evaluations that focus mainly on feature mapping, code-generation correctness, or single-shot task completion [10], [13], [23]. Third, at the practical level, it offers a contextual profile of how different AI IDE agents orchestrate planning, tool use, recovery, verification, and human intervention in an end-to-end learning-media development task. The intended contribution is therefore not to declare a universally superior AI IDE agent, but to provide a replicable process-oriented

evaluation model for mapping workflow trade-offs among rapidly evolving agentic development environments.

## II. METHODOLOGY

Evaluating AI IDE agents requires a methodology that captures not only final outputs but also the workflow processes through which those outputs are produced [13], [24]. To address this need, the present study adopts a descriptive comparative approach that combines quantitative metrics with qualitative evidence from agent interaction logs and internal artifacts [24].

### A. Research Design

This study employed a descriptive comparative evaluation design combining quantitative and qualitative analysis [25]. Four AI IDE agents--Cursor, Windsurf, Trae, and Antigravity--were tested across the same five development stages in sequence under identical scenarios and procedures, with one workflow run per stage and no repeated trials. This design was intended to characterize observable workflow differences within a controlled benchmark scenario rather than to estimate population-level effects. A single project scenario was used to keep the task context, domain requirements, source materials, and completion criteria comparable across agents, allowing the analysis to focus on differences in workflow behavior rather than differences in task selection. Accordingly, the findings are interpreted as descriptive comparative profiles under the observed test conditions rather than as a basis for statistical inference.

### B. Research Objects and Benchmark Scenario

The four AI IDE agents selected for this study--Cursor, Windsurf, Trae, and Antigravity--were justified in the Introduction as representing a range of contemporary software-agent architectures. For inclusion in this evaluation, an AI IDE agent was defined as an IDE-integrated or IDE-like software-development platform that embeds an AI agent capable of translating natural-language development goals into workspace-level actions. Following prior descriptions of agentic programming and software-engineering agents, such systems are expected to support planning or task decomposition, file or code modification, tool interaction such as terminal commands, dependency installation, compilation or testing, and iterative feedback-based recovery during task execution [5], [9], [13]. Tools that only provide passive code completion or isolated chat-based code suggestions without direct workspace-level execution capability were therefore outside the scope of this study.

The technical configuration specifications of each system, including agent mode, AI backend model, and version at the time of testing, are presented in Table 1.

TABLE I
TABLE 1. SPECIFICATION OF TESTED AGENTIC IDE CONFIGURATIONS

| Agentic IDE | Model Configuration During Testing | Agentic IDE Version |
|---|---|---|
| Cursor | Auto | 2.6 |
| Windsurf | Claude Opus 4.6 | 1.9566.11 |
| Trae | Auto | 3.5.13 |
| Antigravity | Claude Opus 4.6 | 1.20.5 |

The benchmark scenario involved the development of Next-Gen SPLDV, an interactive web-based learning platform on Sistem Persamaan Linear Dua Variabel (SPLDV) for ninth-grade junior high school students [26], [27]. The scenario was divided into five progressive stages representing full-stack development tasks with different complexity sources, from project scaffolding to interactive feature implementation with database integration. Stage complexity was defined by the number of required artifacts, the degree of cross-file integration, the presence of external source material, mathematical rendering or computation requirements, database/schema configuration, and the need for verification through build or visual checks. Each AI IDE agent was evaluated as a complete usage package that included its interface, agent mode, and default AI backend model.

1) *Stage 1 (Project Initialization and Interface Scaffolding)*: Initializing a Next.js 15 project with TypeScript strict mode, Tailwind CSS integration, base layout structure, reusable UI components, and dark mode toggle. The task required the agent to configure the project environment, organize a modular folder structure, build UI-kit components such as Button, Card, and Container, implement structural components such as Navbar and Footer, add responsive hamburger-menu behavior, persist dark/light mode preferences through localStorage, and verify that the layout rendered without errors. This stage tested the agent's ability to scaffold a project architecture, configure the development environment, and establish reusable interface foundations.

2) *Stage 2 (Landing Page Content Development)*: Developing informative landing page content on SPLDV topics along with structured UI components. The task required the agent to read the RPP source file, extract learning objectives and triggering questions, separate the extracted content into a typed content module, build a Server Component landing page, and implement interactive question tabs or toggles as a separate Client Component. This stage tested the agent's ability to transform educational source material into a structured web interface while maintaining content organization, server-client component boundaries, responsive layout, and design consistency.

3) *Stage 3 (Learning Content Page with Mathematical Rendering)*: Building a learning content page derived from an external document (PDF/RPP), integrating KaTeX/LaTeX for mathematical notation, and implementing subchapter navigation. The task required the agent to install and

configure KaTeX, model four SPLDV solution methods in a typed data structure, create a reusable LaTeX renderer, render mathematical expressions correctly, and combine a Server Component page with Client Component tab navigation. This stage tested the agent's ability to process external source materials, represent mathematical content in code, and coordinate dependency setup, typed content modeling, mathematical rendering, and interactive navigation.

4) *Stage 4 (Exercise Feature with Database Integration)*: Implementing an interactive quiz feature with SPLDV logic validation, Prisma ORM setup, SQLite database configuration, and schema migration. The task required the agent to initialize Prisma, define and migrate a quiz-result schema, generate the Prisma client, create a reusable Prisma singleton, implement a Server Action for saving quiz results, and integrate these backend elements with a Client Component quiz flow involving student input, answer selection, real-time feedback, scoring, and result persistence. This stage tested full-stack integration capability and the agent's handling of dependency management, database configuration, server-client interaction, validation logic, and build verification.

5) *Stage 5 (Interactive Game Feature Development)*: Developing an interactive mini-game ("Tebak Titik Potong") involving Cartesian coordinate identification, Cramer's Rule computation, and SVG-based graph visualization. The task required the agent to implement mathematical utility functions for generating valid pairs of linear equations, computing intersection points, deriving line coordinates for SVG rendering, validating user answers, rendering a Cartesian grid with axis labels and equation lines, and managing game state, scoring, rounds, and feedback. This stage tested the agent's mathematical logic implementation, algorithmic problem generation, graphics manipulation, interactive state handling, and dynamic visual feedback.

Each AI IDE agent was evaluated as a complete usage package that included its interface, agent mode, and default AI backend model. This approach was chosen because, in practice, users interact with AI IDE agents as unified systems rather than with isolated components; the methodological consequences of this choice are discussed in the Validity and Limitations section.

Together, these stages formed a progressive benchmark in which task demands increased from environment setup and reusable UI scaffolding to educational-content transformation, mathematical rendering, database-backed interactivity, and dynamic mathematical visualization. This staged structure allowed the evaluation to observe whether each agent's workflow remained consistent as complexity shifted across front-end, full-stack, and mathematical-interactive development tasks.

### C. *Variables and Metrics*

To maintain comparability across AI IDE agents, every stage was delivered through standardized prompts containing the task objective, context, technical specifications, completion criteria, and execution instructions. This standardization was intended to reduce variation in instruction interpretation across systems and aligns with findings that prompt structure and system instructions affect the consistency and reliability of outputs in LLM-based tasks [6], [28].

In this study, the independent variable was the type of AI IDE agent used (Cursor, Windsurf, Trae, and Antigravity), whereas the dependent variables were operationalized into four core dimensions of agentic maturity--task decomposition, tool-use effectiveness, autonomous recovery capability, and human intervention cost--along with time completion efficiency as an additional performance indicator. Each dimension was measured through one or more quantitative metrics calculated from the agent's interaction logs during each run.

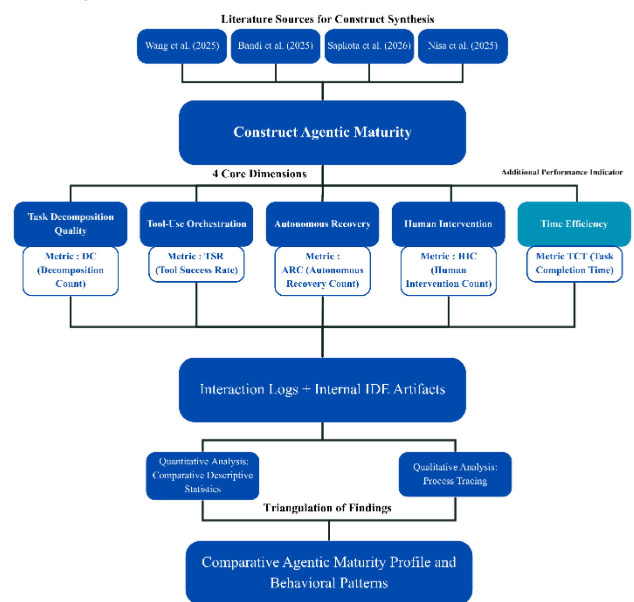


Figure 1. Conceptual Framework of Research Variables

Following prior work that emphasizes planning, tool use, error recovery, and human feedback as key dimensions of agentic software-engineering systems [5], [13], this study operationalized agentic workflow capability into five process-oriented metrics: Decomposition Count (DC), Tool Success Rate (TSR), Autonomous Recovery Count (ARC), Human Intervention Count (HIC), and Task Completion Time (TCT). Because the evaluation focused on workflow behavior rather than only final code output, these metrics were coded from interaction logs and supplementary internal artifacts using a process-tracing approach [25], [29]. To support consistent and replicable assessment across AI IDE agents and benchmark stages, each metric is described below in terms of its conceptual meaning, operational definition, unit of measurement, data source, calculation rule, and interpretation direction.

*1) Task Decomposition*: The task decomposition dimension refers to the agent's ability to translate a high-level objective into smaller, ordered, and executable steps. In this study, this dimension was measured using Decomposition Count (DC). Operationally, DC was defined as the number of explicit steps, sub-steps, or action plans produced by the agent during task completion. The unit of measurement was a count, expressed as a frequency, and the data source consisted of interaction logs and generated planning artifacts such as plan files or implementation documents. As a calculation rule, each explicitly stated planned item was counted once. In terms of interpretation direction, higher DC values indicate more explicit task decomposition, although a higher count reflects planning externalization rather than planning quality in itself. As an illustration of the coding procedure, when an agent's plan file lists fourteen enumerated entries such as "Step 1: initialize project, Step 2: configure Tailwind, Step 3: create base layout", the coder records DC = 14 for that stage; when no separate plan file exists, the same rule is applied to identifiable planning statements in the agent's interaction log.

*2) Tool Use*: The tool use dimension represents the agent's ability to employ available tools to support task completion. In this study, this dimension was measured using three complementary metrics: Tool Attempt, Tool Success, and Tool Success Rate (TSR). Operationally, Tool Attempt was defined as the number of observable tool invocations performed by the agent, while Tool Success was defined as the number of those invocations whose outputs were usable for the next workflow step; TSR was derived from these two counts. Tool Attempt and Tool Success were measured as counts, TSR was expressed in percent, and the data source for all three metrics consisted of tool-use records and execution traces in the interaction logs. As a calculation rule, TSR = (Tool Success / Tool Attempt) x 100. In terms of interpretation direction, higher TSR values indicate more effective tool use, but should be read together with tool-use volume to avoid mistaking a small number of cautious invocations for stronger orchestration. As an illustration of the coding procedure, when the interaction log shows ten distinct tool invocations during a stage and eight of those invocations produced outputs used as input to the next workflow step, the coder records Tool Attempt = 10, Tool Success = 8, and TSR = 80%; failed invocations such as syntax errors, exit code 1, or unusable outputs are counted toward Tool Attempt but not Tool Success.

*3) Error Recovery*: The error recovery dimension refers to the agent's ability to respond to failures, errors, or obstacles encountered during task execution. In this study, this dimension was measured using Autonomous Recovery Count (ARC). Operationally, ARC was defined as the number of self-recovery attempts in which the agent encountered an error or failure and then initiated corrective action — such as editing files, re-running commands, revising inputs, or adjusting configurations — without receiving new instructions from the operator. The unit of measurement was a count, expressed as a frequency, and the data source consisted of error events and interaction logs, including failed commands, runtime errors, configuration conflicts, and corresponding agent-initiated corrections. As a calculation rule, each error-response cycle that proceeded without new operator instruction was counted as one ARC instance. In terms of interpretation direction, higher ARC values indicate more active autonomous recovery, but may also reflect repeated failure-recovery cycles, so ARC is read together with TSR and HIC rather than in isolation. As an illustration of the coding procedure, when the log shows the agent receiving an error from a failed command and then issuing a corrected command on its own without any new operator input, the coder records one ARC instance; each subsequent autonomous retry triggered by a recurring error is counted as an additional ARC instance.

*4) Human Intervention:* The human intervention dimension represents the extent to which human assistance was still required during the agent's task execution process. In this study, this dimension was measured using Human Intervention Count (HIC). Operationally, HIC was defined as the number of operator inputs delivered after the original standardized prompt that were necessary to unblock continuation, correct a blocking issue, or enable task completion. Qualifying inputs included clarifications beyond the original prompt, corrections of agent actions, reissued instructions when the agent became stuck, direct debugging guidance, and resolutions of environment or blocking issues that the agent could not handle autonomously. To reduce evaluator subjectivity, routine observation, final acceptance checking, documentation activities, and operator inputs caused by external issues unrelated to agent capability were excluded. The unit of measurement was a count, expressed as a frequency, and the data source consisted of intervention notes and interaction logs. As a calculation rule, each qualifying operator input was counted as one HIC instance. In terms of interpretation direction, lower HIC values indicate a higher degree of agent autonomy, although the metric remains partly sensitive to evaluator judgment. As an illustration of the coding procedure, when the operator types a clarification such as "use SQLite as the database, not PostgreSQL" to unblock an agent that had stalled on a configuration choice, the coder records one HIC instance, whereas operator inputs caused by external issues unrelated to the agent's behavior are excluded.

*5) Time Efficiency*: The time efficiency dimension reflects the temporal efficiency of the agent in completing a task from start to finish. In this study, this dimension was measured using Task Completion Time (TCT). Operationally, TCT was defined as the total time required to complete a single experimental task, including time spent during human intervention, since this represents the actual completion duration under real-world usage conditions. The unit of measurement was minutes, and the data source consisted of stage timestamps recorded at prompt delivery and at verified

stage completion. As a calculation rule, TCT was computed as the elapsed time from the delivery of the stage prompt to the verification of the final output. In terms of interpretation direction, lower TCT values indicate higher time efficiency, but should be read together with ARC and HIC because a shorter completion time does not necessarily imply lower recovery or intervention cost. As an illustration of the coding procedure, when the stage prompt is delivered at 14:00 and the final output is verified as complete at 14:13, the coder records TCT = 13 minutes for that stage; intervention time within this interval is included, since it is part of the actual completion duration.

The same conceptual definitions, operational definitions, units, data sources, calculation rules, and interpretation directions were applied uniformly to all evaluated AI IDE agents and benchmark stages. The scoring procedure was conducted through semi-manual log coding rather than through a Likert-type rubric or fully automated scoring, because several workflow events, such as decomposition steps, autonomous recovery cycles, intervention boundaries, and successful tool use, required contextual interpretation from the recorded agent behavior. This choice is consistent with prior work on agentic software-engineering evaluation [5], [13] and process-oriented case-study analysis [25], [29].

To support consistent coding across agents and stages, the raw process evidence for each run was organized into a structured archive. For every AI IDE agent and benchmark stage, the archive grouped the recorded planning steps, the chronological tool-use trace with success or failure status, the autonomous recovery events, the operator intervention events, the start and end timestamps, and brief qualitative notes on observed errors and outcomes. The metric values used in this study were derived directly from this structured archive, so that each reported score could be traced back to a specific set of recorded events for the corresponding agent and stage rather than to subjective overall impressions.

Stage-level scores were then aggregated to produce the overall profile for each AI IDE agent. Mean DC was calculated as the average DC across the five benchmark stages. Total Tool Attempt, Total Tool Success, Total ARC, Total HIC, and Total TCT were calculated by summing the corresponding stage-level values. Overall TSR was calculated by dividing Total Tool Success by Total Tool Attempt and multiplying the result by 100. The coded values were checked against the detailed log sheets, qualitative error notes, intervention records, recovery-event notes, and final verification evidence to ensure that each score corresponded to observable workflow evidence.

### *D. Experimental Procedure*

For each run, the procedure followed four sequential stages: preparation, execution, recording, and verification. All stages were conducted by a single operator (the researcher) to maintain treatment consistency across sessions.

During the preparation stage, testing consistency was maintained by using the same benchmark scenario, stage sequence, source materials, project constraints, completion criteria, and verification expectations for all AI IDE agents. The standardized stage prompts contained the same categories of information across agents, namely the stage objective, project context, technical specifications, required artifacts, definition of done, and execution instructions. Each agent was run in a separate fresh project directory to avoid carry-over artifacts across sessions, and configuration details, including agent name, active agent mode, version, and AI backend model, were recorded before execution. The same benchmark resources were provided across runs, including the SPLDV source material, project specification files, and stage-specific requirements.

During the execution stage, the prompt content was adjusted only to match the current stage, not the agent being tested. No additional guidance was provided beyond the predefined instructions unless the agent repeatedly stopped producing new actions or encountered a critical error that blocked the build process or stage continuation. Operator intervention followed the same predefined rules across agents. During the recording stage, relevant events were documented in real time during the run, including human interventions (HIC) with brief justifications, autonomous recovery attempts (ARC), tool use and success status, and timestamps for TCT calculation. During the verification stage, the final output was checked against the same stage-specific functional specifications and completion criteria to ensure alignment of the generated artifacts. Platform-specific interface, logging, and backend differences were unavoidable and are therefore treated as part of the validity limitations.

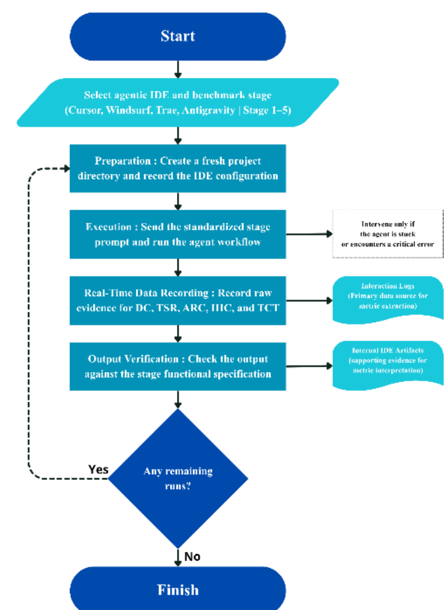


Figure 2. Experimental Procedure Flowchart

Primary data were obtained from interaction logs and internal artifacts. Exported Markdown logs were available for Cursor, Windsurf, and Antigravity, while Trae logs were

documented manually through copy-paste because no built-in export feature was available. To reduce the effect of differences in logging granularity, the study collected supplementary internal artifacts from each platform and used cross-source triangulation, so that the process tracing strategy could adequately maintain decision and execution traces. These artifacts included plan files (Cursor) [10], internal prompts and snapshots (Windsurf) [10], turn-level git snapshots (Trae) [18], and implementation plans and walkthrough documents (Antigravity) [19]. Because logging granularity differed across platforms, the archived artifacts were used to triangulate each run rather than to assume identical log formats across agents.

### E. Data Analysis

The data were analyzed through complementary quantitative and qualitative approaches. Quantitatively, the study used descriptive comparative statistics rather than inferential significance testing. For each metric, values were calculated per AI IDE agent per benchmark stage and then aggregated into cross-stage profiles. The quantitative comparison included stage-level values, cross-stage totals or means, percentage-based indicators such as TSR, and rank-based interpretation across agents. These descriptive statistics were used to identify performance gaps, workflow trade-offs, and stage-specific variations across task decomposition, tool use, error recovery, human intervention, and time efficiency.

Inferential significance testing was not applied because the experimental design used one workflow run per stage for each agent, while the five stages represented heterogeneous development tasks rather than repeated independent trials of the same task. Under this design, p-values or conventional statistical significance claims could give a misleading impression of generalizability. Therefore, the observed differences are interpreted as descriptive quantitative differences within the defined benchmark conditions, not as statistically significant evidence of inherent superiority among agents.

Qualitatively, process tracing of interaction logs and internal artifacts from each AI IDE agent was conducted to triangulate the evidence [25], [29]. The qualitative analysis focused on four workflow markers: planning evidence, execution evidence, verification evidence, and recovery or intervention evidence. From this process, recurring error patterns across systems, recovery strategies employed by each agent, and distinctive or anomalous tool-use behaviors were identified. Quantitative and qualitative findings were then triangulated: metrics were used to map performance differences, while qualitative evidence from logs provided interpretive explanations of the mechanisms underlying those differences.

### F. Validity and Limitations

Internal validity was supported through prompt standardization, separate fresh project directories for each agent, and consistent intervention rules that limited operator involvement to stuck conditions or critical errors, while construct validity was maintained by aligning all metrics with agentic maturity dimensions from prior literature. However, several threats to validity remained, including differences in underlying AI backend models, possible backend and feature updates during or after the experiment, operator learning effects, variations in logging granularity, the absence of pilot runs, the lack of counterbalancing in execution order, and the use of only a single run for each stage [30].

A further limitation concerns the rapidly evolving nature of AI IDE agents. Because these systems are commercial and continuously updated, their backend models, agent modes, tool interfaces, default system prompts, context-management behavior, and recovery mechanisms may change after the evaluation period. Therefore, the reported results should be understood as a temporal snapshot of the tested versions and configurations rather than as fixed properties of the tools. Reporting the model and IDE versions improves transparency and reproducibility, but it does not remove the risk of temporal validity threats. Repeated evaluations after major model or product updates are needed to determine whether the observed workflow patterns remain stable over time.

Although procedural standardization, internal artifact collection, and quantitative-qualitative triangulation were applied to reduce these risks, the use of a single SPLDV web application scenario limits generalization beyond similar educational web-development contexts. The findings therefore should not be assumed to represent agent behavior in substantially different software domains, such as security-sensitive systems, large-scale enterprise applications, backend API services, maintenance/debugging tasks, or non-educational software.

The most defensible generalization from this study is therefore analytical rather than statistical. The specific performance profiles of each AI IDE agent may transfer most directly to web-application tasks with similar characteristics, such as user-interface construction, educational content organization, client-side interactivity, lightweight database integration, and framework-level configuration. In contrast, domains involving security-critical requirements, large-scale distributed architecture, legacy maintenance, performance optimization, native mobile development, or low-level systems programming may expose different failure modes and require different forms of verification. Nevertheless, the evaluation dimensions used in this study--planning externalization, tool-use effectiveness, recovery behavior, human intervention need, and time efficiency--can serve as a reusable lens for comparing agentic workflows across other software-development domains. Future work should extend the benchmark to multiple project scenarios with different architectural, domain, and verification characteristics to test whether the observed workflow patterns remain stable across contexts.

## III. Result and Discussion.

### A. Overall Performance Profile

To provide a cross-stage summary of agent performance, the overall results of the four AI IDE agents were aggregated across the five benchmark stages using seven indicators, namely mean task decomposition (DC), total tool attempt, total tool success, overall tool-use effectiveness (TSR), total autonomous recovery capability (ARC), total human intervention cost (HIC), and total task completion time (TCT).

TABLE II
OVERALL PERFORMANCE PROFILE OF THE FOUR AI IDE AGENTS

| AI IDE Agent | Mean DC | Total Tool Attempt | Total Tool Success | Overall TSR | Total ARC | Total HIC | Total TCT |
|---|---|---|---|---|---|---|---|
| Antigravity | 7.4 | 151 | 145 | 96.00% | 8 | 3 | 98 |
| Cursor | 6.0 | 101 | 92 | 91.10% | 13 | 3 | 125 |
| Trae | 7.2 | 134 | 118 | 88.10% | 16 | 4 | 103 |
| Windsurf | 6.6 | 204 | 184 | 90.20% | 15 | 6 | 137 |

As shown in Table 2, the four AI IDE agents did not form a simple linear ranking. Instead, each system showed a different workflow trade-off profile under the observed benchmark conditions.

Antigravity appeared relatively stable within this specific benchmark, combining the highest TSR at 96.00%, the lowest total ARC at 8, a low HIC of 3, and the shortest total TCT at 98 minutes. However, this profile should be interpreted as contextual stability within the tested scenario rather than as evidence of general superiority across software-development domains. Cursor showed a more selective workflow profile, with the lowest tool-use volume at 101 attempts, a solid TSR of 91.10%, and a low HIC of 3, although its total completion time was longer than Antigravity and Trae.

Trae showed a speed-oriented profile, with the second shortest total TCT at 103 minutes and a relatively high mean DC of 7.2, but its lower overall TSR of 88.10% and higher ARC of 16 indicate greater vulnerability during failure-prone stages. Windsurf showed the most exploratory profile, with the highest tool-use volume at 204 attempts and 184 successful executions, but also the highest HIC at 6 and the longest total TCT at 137 minutes. Overall, these findings suggest that agentic workflow capability is better understood as a set of trade-offs among planning explicitness, tool-use efficiency, recovery burden, intervention need, and time efficiency.

### B. Task Decomposition (DC)

As shown in Figure 3, decomposition count varied across agents and stages. In Stage 1, Trae recorded the highest DC with 14 steps, followed by Antigravity with 12, Windsurf with 10, and Cursor with 8.

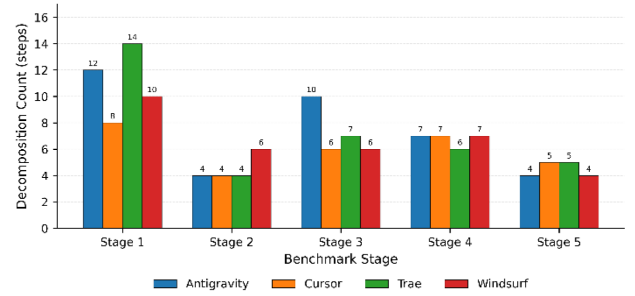


Figure 3. Decomposition Count (DC) across Benchmark Stages

This indicates that project initialization and interface scaffolding encouraged agents to externalize broader planning structures. By contrast, Stages 2 and 5 showed generally lower DC values across all agents, suggesting that not all tasks required the same degree of explicit decomposition. In Stage 4, DC values converged within a narrow range of 6 to 7 steps, indicating a more similar decomposition pattern across agents when handling database-related tasks. Therefore, DC is better interpreted as a measure of planning externalization rather than as a direct measure of planning quality.

### C. Tool-Use Effectiveness (TSR)

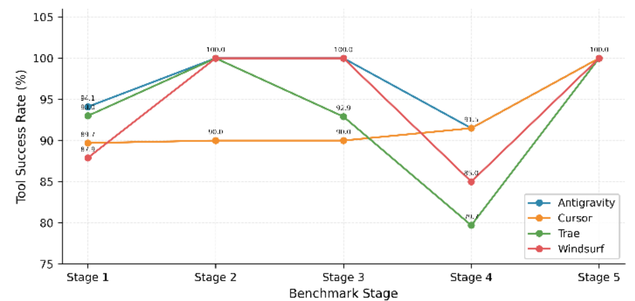


Figure 4. Tool Success Rate (TSR) across Benchmark Stages

As shown in Figure 4, tool-use effectiveness was strongly influenced by stage demands. All agents achieved a TSR of 100 percent in Stage 5, indicating that the recorded tool executions in this stage were successful within the observed run. The clearest decline appeared in Stage 4, which involved database integration using SQLite and Prisma. In this stage, Trae recorded the lowest TSR at 79.7 percent, followed by Windsurf at 85.0 percent, while Antigravity and Cursor each recorded 91.5 percent. This contrast shows that task difficulty was not only related to stage order, but also to the type of technical complexity involved. The interactive game task in Stage 5 required mathematical logic and SVG-based visualization, but it did not generate the same level of dependency, configuration, and database-integration failures observed in Stage 4.

### D. Autonomous Recovery Capability (ARC)

As shown in Figure 5, autonomous recovery capability also varied by stage, with the highest ARC values appearing in Stage 4 for all agents. In this stage, Trae recorded 10

recovery attempts, followed by Windsurf with 8, while Antigravity and Cursor each recorded 5.

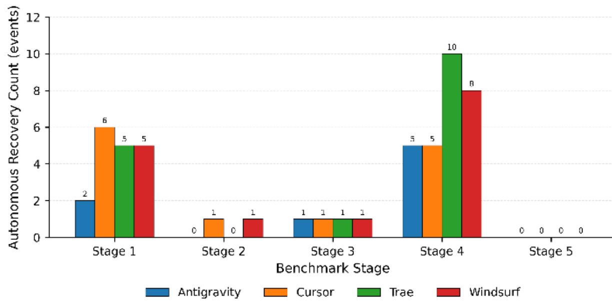


Figure 5. Autonomous Recovery Count (ARC) across Benchmark Stages

Within this benchmark, this pattern indicates that database integration generated the greatest observed need for self-correction across systems. By contrast, all agents recorded an ARC of 0 in Stage 5, showing that no autonomous recovery was required in the interactive game task. A high ARC should therefore be interpreted carefully: it may indicate active self-correction, but it may also reflect repeated failure-recovery cycles.

### E. *Human Intervention Cost (HIC)*

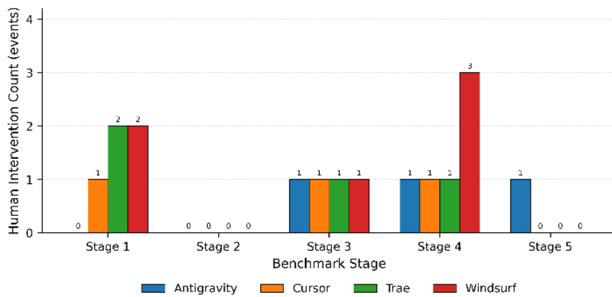


Figure 6. Human Intervention Count (HIC) across Benchmark Stages

As shown in Figure 6, human intervention cost varied across agents and stages. Overall, Antigravity and Cursor recorded the lowest total HIC values, both at 3, followed by Trae with 4 and Windsurf with 6. No agent required human intervention in Stage 2 during the observed runs, indicating that this stage was completed without operator support across all systems in this benchmark. By contrast, Stage 4 showed the greatest intervention demand, particularly for Windsurf, which recorded 3 interventions. These findings suggest that HIC remained low in simpler or more self-contained stages but increased when agents encountered more demanding implementation and integration tasks.

### F. *Time Completion Efficiency (TCT)*

As shown in Figure 7, task completion time differed not only across agents but also according to stage demands. In total, Antigravity recorded the shortest completion time at 98 minutes, followed by Trae at 103 minutes, Cursor at 125 minutes, and Windsurf at 137 minutes.

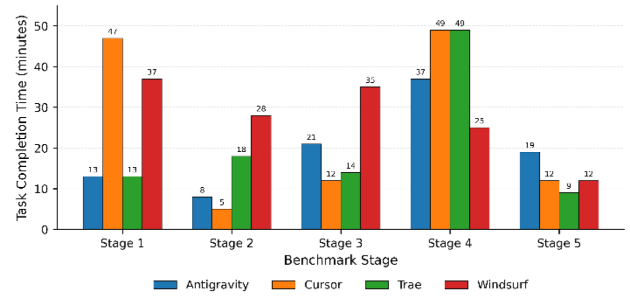


Figure 7. Task Completion Time (TCT) across Benchmark Stages

In Stage 1, Antigravity and Trae each completed the task in 13 minutes, while Cursor and Windsurf required 47 and 37 minutes, respectively. The longest completion times appeared in Stage 4, where Cursor and Trae each required 49 minutes, Antigravity required 37 minutes, and Windsurf required 25 minutes. The fact that Windsurf completed Stage 4 faster despite higher intervention and recovery burden shows that TCT should not be interpreted independently. A shorter completion time may reflect faster iteration, but it does not necessarily indicate lower recovery cost or lower intervention need.

### G. *Agent-Level Workflow Profiles*

The decomposition, tool-use, recovery, intervention, and time patterns suggest different workflow styles across systems. Antigravity and Trae tended to externalize more detailed initial planning in broader task stages, whereas Cursor and Windsurf were more concise. Similarly, the contrast between tool-use volume and TSR indicates that greater tool activity did not automatically correspond to better execution accuracy. These patterns reinforce the importance of reading TSR, ARC, HIC, and TCT together as a connected workflow profile rather than treating any single metric as a standalone indicator of autonomy.

To make the workflow comparison more systematic, the qualitative log analysis categorized evidence into four workflow markers: planning evidence, execution evidence, verification evidence, and recovery or intervention evidence. Planning evidence included explicit plans, task files, implementation notes, or identifiable planning statements in the logs. Execution evidence included file creation or modification, terminal commands, tool invocations, dependency installation, migration commands, and browser or visual checks. Verification evidence included TypeScript checks, build attempts, database inspection, screenshots, walkthrough files, or explicit completion checks against the stage definition of done. Recovery or intervention evidence included failed commands, runtime errors, configuration conflicts, autonomous correction attempts, and operator inputs required to unblock the workflow. The plan-execute-verify interpretation was therefore derived by comparing how these workflow markers appeared across agents and stages rather than by relying only on a narrative reading of the logs.

Using these workflow markers, qualitative evidence from interaction logs and internal artifacts was used to clarify the

behavioral differences observed among systems. Table 3 summarizes the workflow markers observed for each AI IDE agent across the five benchmark stages, with each cell reporting artifact-level or behavioral evidence drawn from the recorded logs and internal artifacts. The per-agent paragraphs below then expand on these markers with concrete recovery details, especially around the Stage 4 Prisma v7 breaking change, providing traceable log-based support for interpreting workflow differences rather than relying only on narrative impression.

| AI IDE Agent | Planning Evidence | Execution Evidence | Verification Evidence | Recovery/Intervention Evidence |
|---|---|---|---|---|
| Antigravity | Explicit `implementation_plan.md` and `task.md` with step-by-step breakdowns and verification checkpoints before execution. | File creation/modification, terminal commands, and targeted dependency installation (e.g., `@prisma/adapter-better-sqlite3`) configured via `PrismaBetterSqlite3` factory. | `npx tsc --noEmit` after major file creation, browser screenshots, and post-execution `walkthrough.md` documenting completed work. | Focused recovery on the Prisma v7 breaking change (5 attempts via explicit adapter configuration); total ARC 8 and HIC 3 across the five stages. |
| Cursor | Compact `.plan.md` files with concise task lists; occasional delegation to a background subagent observed in transcripts. | Lowest tool-use volume (101 attempts) with selective file edits and terminal commands. | TypeScript checks at the end of implementation cycles and `npx prisma studio` for database verification, with less frequent intermediate checks. | Selective recovery on Prisma v7 (5 attempts via a type-casting workaround that bypassed strict constructor requirements); total ARC 13 and HIC 3. |
| Trae | Implicit planning through conversation; no explicit plan files. Progress tracked via turn-level git snapshots and patch files. | Code edits accompanied by incremental compilation feedback; per-turn git snapshots captured intermediate working states. | Verification largely relied on incremental code compilation and git snapshots rather than dedicated build/type checks; less systematic verification routine. | Most varied recovery on Prisma v7 (10 attempts across empty-object constructors, datasource options, and adapter exploration before succeeding via type assertion); total ARC 16 and HIC 4. |
| Windsurf | `plan.md` files with moderate detail, complemented by `.snapshot` files capturing key component states. | Highest tool-use volume (204 attempts) with repeated build commands and broad exploratory edits. | Build commands used as reactive verification; repeated errors triggered additional recovery cycles rather than upfront type or schema checks. | Extended recovery chains; eventually downgraded to Prisma v6 to restore the built-in SQLite driver (8 attempts in Stage 4); total ARC 15 and HIC 6. |

The markers in Table 3 are not intended as a quality ranking; they show that each agent emphasized different combinations of planning externalization, tool-use selectivity, verification routine, and recovery strategy.

1) *Antigravity*: addressed the Prisma v7 breaking change directly through the documented adapter route, treating the migration as an architectural shift to be aligned with rather than a constraint to be worked around (Table 3). This focused recovery was consistent with a broader plan-first style, in which planning artifacts framed each stage, type-check verification followed major edits, and post-execution walkthroughs documented the completed work. As shown in Table 2, the resulting profile combined moderate tool-use volume with the highest execution accuracy among the four agents within the observed benchmark.

2) *Cursor*: handled the same breaking change through a minimalist workaround that bypassed Prisma v7's stricter client construction without introducing additional dependencies (Table 3). This response aligned with its broader profile of compact plan files and occasional subagent delegation. As shown in Table 2, Cursor exhibited the lowest tool-use volume among the four agents while maintaining a comparable success rate, consistent with a selective rather than exploratory orientation.

3) *Windsurf*: showed the most iterative recovery trajectory on the Prisma v7 issue, cycling through several constructor and configuration variants before converging on a working type-assertion fix (Table 3). This iterative behavior aligned with its broader pattern of implicit planning through conversation and turn-level git snapshots rather than dedicated plan files. As shown in Table 2 and Figure 4, the largest drop in execution accuracy was concentrated in the database-integration stage, while the rest of the workflow remained closer to the cross-agent pattern.

4) *Trae*: ultimately stepped back from the Prisma v7 migration by reverting to a previous major version that retained the built-in driver, after several configuration attempts had not stabilized (Table 3). This step-back response

was consistent with a more exploratory trial-and-repair orientation, in which build commands functioned as reactive verification rather than upfront type or schema checks. As shown in Table 2, this orientation coincided with the highest tool-use volume among the four agents within the observed benchmark.

### *H. Cross-System Recovery Patterns and Study Boundaries*

Two common error patterns emerged across multiple systems. First, the Prisma v7 breaking change affected all four AI IDE agents during Stage 4, as Prisma v7 removed built-in database drivers and required explicit adapter configuration. Each system responded differently: Antigravity installed the required adapter, Cursor used type casting, Trae experimented with multiple constructor and configuration variants, and Windsurf reverted to a previous major version. Second, the PowerShell && syntax error appeared when agents attempted to chain npm commands using bash-style syntax. This was observed in Antigravity, Cursor, and Trae during early stages; all corrected by running commands separately. Windsurf did not prominently exhibit this pattern in the analyzed records.

These qualitative patterns provide interpretive context for the quantitative differences observed in ARC, HIC, and TSR. The variation in recovery strategies and planning styles suggests that autonomy is more meaningfully evaluated through multiple workflow dimensions read together rather than through any single metric in isolation.

The quality of the final application output was considered through functional verification at the end of each stage, in which the generated artifacts were checked against the stage-specific completion criteria. However, maintainability, code quality, and security vulnerability were not treated as separate quantitative metrics in this study. This boundary was maintained because the primary objective was to evaluate agentic workflow capability, including planning, tool use, recovery, verification, and intervention needs, rather than to conduct a full software-quality audit. As a result, a workflow that appears efficient may still produce artifacts with different levels of maintainability, readability, technical debt, or security risk that the present design cannot detect.

## IV. CONCLUSION

This study comparatively evaluated the agentic workflow capabilities of Cursor, Windsurf, Trae, and Antigravity through a five-stage benchmark involving the development of the Next-Gen SPLDV educational web application. Within this testing context, the findings map trade-offs across systems: Antigravity appeared relatively more stable within the observed benchmark, whereas Cursor, Trae, and Windsurf showed more context-dependent strengths in selective tool use, partial stage-level efficiency, and exploratory intensity. Accordingly, the answer to RQ1 is positioned as a contextual comparative map rather than the identification of a universally superior system across all dimensions.

In line with this, the answer to RQ2 indicates that the observed performance differences in this benchmark were mainly associated with variations in workflow orchestration. Based on the workflow markers observed in the logs, Antigravity tended to exhibit a more structured plan-execute-verify pattern; Cursor was more minimalist and selective; Windsurf displayed a more aggressive trial-and-repair orientation; and Trae stood out in detailed code-evolution documentation, accompanied by limited indications of cross-stage adjustment. These findings suggest that comparative interpretation of AI IDE agents should be grounded in workflow process—covering planning, tool selection, response to errors, and the need for human intervention—rather than final outputs alone.

From a practical perspective, the selection of AI IDE agents should be aligned with usage priorities and revalidated within real implementation contexts. In this benchmark, Antigravity aligned more closely with workflows emphasizing stable orchestration and lower intervention, while Cursor and Trae remained competitive for more focused tasks, and Windsurf may be more relevant when rapid exploration is prioritized, albeit with the trade-off of higher intervention and recovery effort. Conceptually, the contribution of this study lies in reinforcing the argument that the evaluation of AI IDE agents should integrate process metrics with qualitative evidence of agent behavior so that performance is not reduced to final outputs alone.

At the same time, this conceptual contribution must be interpreted within the limits of the descriptive design used here. The study was limited to one benchmark scenario, one workflow run per stage without replication, and differing configurations and AI backend models across AI IDE agents. The study also did not conduct a dedicated output-quality audit such as maintainability, code-quality scoring, or security-vulnerability assessment; final artifacts were checked primarily against functional completion criteria and workflow evidence. Therefore, the findings should be read as a descriptive comparative portrait under the observed test conditions rather than as cross-context generalization. Future research should extend the range of scenarios, increase run replication, and repeat evaluations after major model or product updates to test the temporal stability of the observed workflow patterns. Future studies should also incorporate dedicated output-quality assessments, such as static code analysis, maintainability scoring, test coverage, and security-vulnerability scanning, alongside workflow-process metrics so that agentic workflow capabilities can be mapped more comprehensively.

## REFERENCES

[1] A. Fan *et al.*, "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)*, IEEE, 2023, pp. 31–53. doi: 10.1109/ICSE-FoSE59343.2023.00008.

[2] Q. Zhang *et al.*, “A survey on large language models for software engineering,” *Science China Information Sciences*, vol. 69, no. 4, 2026, doi: 10.1007/s11432-025-4670-0.

[3] Y. Majdoub and E. Ben Charrada, “Debugging with Open-Source Large Language Models: An Evaluation,” in *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, in ESEM ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 510–516. doi: 10.1145/3674805.3690758.

[4] N. Davila *et al.*, “An Industry Case Study on Adoption of AI-based Programming Assistants,” in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, in ICSE-SEIP ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 92–102. doi: 10.1145/3639477.3643648.

[5] H. Wang, J. Gong, H. Zhang, J. Xu, and Z. Wang, “AI Agentic Programming: A Survey of Techniques, Challenges, and Opportunities,” 2025. [Online]. Available: https://arxiv.org/abs/2508.11126

[6] S. Suri, S. N. Das, K. Singi, K. Dey, V. S. Sharma, and V. Kaulgud, “Software Engineering Using Autonomous Agents: Are We There Yet?,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2023, pp. 1855–1857. doi: 10.1109/ASE56229.2023.00174.

[7] S. Yao *et al.*, “ReAct: Synergizing Reasoning and Acting in Language Models,” 2023. [Online]. Available: https://arxiv.org/abs/2210.03629

[8] T. Schick *et al.*, “Toolformer: Language Models Can Teach Themselves to Use Tools,” 2023. [Online]. Available: https://arxiv.org/abs/2302.04761

[9] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, “Demystifying LLM-Based Software Engineering Agents,” *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 801–824, 2025, doi: 10.1145/3715754.

[10] M. Shrivastava, “A Comparative Featureset Analysis of Agentic IDE Tools,” Jun. 2025, doi: 10.20944/preprints202506.0821.v1.

[11] M. Chen *et al.*, “Evaluating Large Language Models Trained on Code,” *arXiv preprint arXiv:2107.03374*, 2021, [Online]. Available: https://arxiv.org/abs/2107.03374

[12] C. E. Jimenez *et al.*, “SWE-bench: Can Language Models Resolve Real-World GitHub Issues?,” in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024. [Online]. Available: https://arxiv.org/abs/2310.06770

[13] A. Bandi, B. Kongari, R. Naguru, S. Pasnoor, and S. V. Vilipala, “The Rise of Agentic AI: A Review of Definitions, Frameworks, Architectures, Applications, Evaluation Metrics, and Challenges,” *Future Internet*, vol. 17, no. 9, p. 404, Sep. 2025, doi: 10.3390/fi17090404.

[14] R. Peredo, A. Canales, A. Menchaca, and I. Peredo, “Intelligent Web-based education system for adaptive learning,” *Expert Syst. Appl.*, vol. 38, no. 12, pp. 14690–14702, 2011, doi: 10.1016/j.eswa.2011.05.013.

[15] B. Mostefai, T. Boutefara, N. Bousbia, A. Balla, S. Dhelim, and A. Hammia, “Enhancing user experience in e-learning systems: A new user-centric RESTful web services approach,” *Computers in Human Behavior Reports*, vol. 18, p. 100643, 2025, doi: 10.1016/j.chbr.2025.100643.

[16] A. R. Marsa and R. Yunita, “Website Media Pembelajaran Matematika Berbasis Moodle Platform,” *JOISIE (Journal Of Information Systems And Informatics Engineering)*, vol. 3, no. 1, p. 1, 2019, doi: 10.35145/joisie.v3i1.332.

[17] K. M. Lyons, N. G. Lobczowski, J. A. Greene, J. Whitley, and J. E. McLaughlin, “Using a design-based research approach to develop and study a web-based tool to support collaborative learning,” *Comput. Educ.*, vol. 161, p. 104064, 2021, doi: 10.1016/j.compedu.2020.104064.

[18] Trae Inc., “SOLO Builder Documentation,” 2026.

[19] Google LLC, “Antigravity Documentation and Product Materials,” 2026.

[20] U. Nisa, M. Shirazi, M. A. Saip, and M. S. M. Pozi, “Agentic AI: The age of reasoning—A review,” *Journal of Automation and Intelligence*, vol. 5, no. 1, pp. 69–89, 2026, doi: 10.1016/j.jai.2025.08.003.

[21] R. Sapkota, K. I. Roumeliotis, and M. Karkee, “AI Agents vs. Agentic AI: A Conceptual taxonomy, applications and challenges,” *Information Fusion*, vol. 126, p. 103599, 2026, doi: 10.1016/j.inffus.2025.103599.

[22] L. Wang *et al.*, “A survey on large language model based autonomous agents,” *Front. Comput. Sci.*, vol. 18, no. 6, 2024, doi: 10.1007/s11704-024-40231-1.

[23] B. Kitchenham *et al.*, “Robust Statistical Methods for Empirical Software Engineering,” *Empir. Softw. Eng.*, vol. 22, no. 2, pp. 579–630, 2017, doi: 10.1007/s10664-016-9437-5.

[24] A. Velasco, “Beyond Accuracy: Evaluating Source Code Capabilities in Large Language Models for Software Engineering,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, in ICSE-Companion ’24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 162–164. doi: 10.1145/3639478.3639815.

[25] P. Runeson and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” *Empir. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, 2008, doi: 10.1007/s10664-008-9102-8.

[26] R. H. Pratiwi, R. A. Fajri, and Hasbullah, “Effectiveness of Implementing Open Source-Based E-Learning Media ‘Mathematics Laboratory: SPLDV’ on Mathematical Problem Solving Skills of MTs Students in Tangerang Regency,” *Jurnal Penelitian Pendidikan IPA*, vol. 11, no. 3, pp. 484–493, 2025, doi: 10.29303/jppipa.v11i3.9901.

[27] W. Villegas-Ch, D. Buenano-Fernandez, A. M. Navarro, and A. Mera-Navarrete, “Adaptive intelligent tutoring systems for STEM education: analysis of the learning impact and effectiveness of personalized feedback,” *Smart Learning Environments*, vol. 12, no. 1, Jun. 2025, doi: 10.1186/s40561-025-00389-y.

[28] M. Rizqullah and E. Albassam, “Large Language Model Selection for Test-Driven Prompt Android iOS Development,” *International Journal of Interactive Mobile Technologies (iJIM)*, vol. 20, no. 03, Feb. 2026, doi: 10.3991/ijim.v20i03.59861.

[29] C. Wohlin, “Case Study Research in Software Engineering—It is a Case, and it is a Study, but is it a Case Study?,” *Inf. Softw. Technol.*, vol. 133, p. 106514, 2021, doi: https://doi.org/10.1016/j.infsof.2021.106514.

[30] R. Verdecchia, E. Engström, P. Lago, P. Runeson, and Q. Song, “Threats to validity in software engineering research: A critical reflection,” *Inf. Softw. Technol.*, vol. 164, p. 107329, 2023, doi: https://doi.org/10.1016/j.infsof.2023.107329.