Benchmarking Agentic Frameworks for Incremental Software Evolution

Vitor Linhares
State University of Ceará
Fortaleza, Brazil
vitor.fontenele@aluno.uece.br

Anderson Gomes
State University of Ceará
Fortaleza, Brazil
anderson.martins@aluno.uece.br

Ingrid Vieira
State University of Ceará
Fortaleza, Brazil
ing.vieira@aluno.uece.br

Paulo Maia
State University of Ceará
Fortaleza, Brazil
pauloh.maia@uece.br

Abstract

Agentic frameworks are increasingly used in software engineering, but comparing them remains difficult due to differences in goals, output formats, and evaluation practices. This paper extends the evaluation of Business Autonomous Entities (BAEs), an approach for incremental software evolution, through a controlled benchmark against five agentic software engineering frameworks: ChatDev, GitHub Spec Kit, MetaGPT, AutoGen, and Self-Organized Agents (SOA). All frameworks evolve the same Student–Course system through six incremental changes, with 50 executions per framework and 300 runs in total. We analyze execution time, token usage, monetary cost, API calls, cached tokens, and *sounds good*, a binary structural sanity metric for checking whether generated artifacts satisfy a minimal set of prompt requirements. Results show that no framework dominates all dimensions. ChatDev obtains the highest *sounds good* rate, but at higher cost and execution time. BAEs and AutoGen achieve nearly the same *sounds good* rate while remaining faster and cheaper, and BAEs show a more stable cost profile. These findings suggest that agentic software engineering frameworks should be evaluated not only by raw efficiency, but also by artifact usability, stability, and support for incremental evolution.

Keywords

Agentic Frameworks, Benchmark, Business Autonomous Entities, Software Evolution, LLMs

1 Introduction

Artificial Intelligence (AI), Large Language Models (LLMs), and agentic systems are increasingly reshaping software engineering [7]. These technologies have been used to support several development activities, including code generation, testing, debugging, maintenance, and requirements refinement [4]. In this context, several agentic software engineering frameworks have emerged to connect LLMs with software development workflows. Some of them coordinate multiple agents, roles, tools, or structured workflows, while others are closer to prompt-driven generation with limited memory across tasks [8, 13]. Examples include ChatDev [13], MetaGPT [8], GitHub Spec Kit [5], AutoGen [15], and Self-Organized Agents (SOA) [9]. These frameworks, however, do not necessarily follow the same assumptions. Some are closer to conversational software generation, others to specification-driven development, and others to general multi-agent orchestration. This diversity is useful, but it also makes comparison harder.

In this paper, we focus on frameworks that use LLMs as part of a software development workflow, rather than on end-user coding assistants such as Claude Code, Codex, Antigravity, or similar products. This distinction is important because our goal is not to compare interactive developer tools, but to evaluate frameworks that can be executed as experimental subjects under a controlled software evolution task. Despite their progress, agentic frameworks for software engineering are still difficult to evaluate. This is not a limitation of a single framework, but a broader challenge in the area [12]. As in software engineering more generally, there is no silver bullet [2]: different frameworks optimize for different goals, expose different interfaces, produce different artifacts, and report different metrics, even when their final goal is similar, namely to support the production of useful software artifacts. As a result, direct comparison is often fragile. A framework may appear strong under the metrics used in its own evaluation, while another may optimize for a different dimension. This fragmentation has been discussed in previous work on evaluation metrics for LLM-based multi-agent frameworks [11].

This paper builds on two related observations. First, evaluating agentic software engineering frameworks requires more than asking whether they can generate code. It also requires understanding how expensive, slow, interaction-heavy, stable, and usable their outputs are under repeated execution. Second, software evolution is different from one-shot generation: real systems evolve incrementally, with their functional content and structure changing over time as new requirements, constraints, and adaptations accumulate [10]. Treating each change as an isolated prompt may be convenient, but it can also discard naming conventions, domain assumptions, and architectural intent. This is especially problematic when the goal is not only to create code, but to evolve a real system while preserving its meaning.

Business Autonomous Entities (BAEs) were proposed to address this problem from an incremental software evolution perspective [6]. A BAE maintains a persistent representation of a business concept and uses it to guide software changes over time. Instead of rebuilding the system from scratch at each new requirement, BAEs aim to preserve domain meaning while delegating implementation tasks to specialized software engineering agents. The original BAEs evaluation provided evidence that this approach can reduce cost,

execution time, token usage, and API calls when compared with a smaller set of baselines [6]. However, that evaluation did not cover a broader set of agentic frameworks, and therefore left open the question of how BAEs behave under a wider cross-framework comparison.

This paper connects the BAE proposal with the metric-oriented perspective introduced in [11]. Rather than proposing a new architecture, we focus on evaluation. Thus, the main contribution of this paper is an extended empirical benchmark of agentic software engineering frameworks. BAEs are included as one of the evaluated frameworks, while their architecture and original evaluation are described in [6]. We benchmark BAEs against five agentic software engineering frameworks: ChatDev, GitHub Spec Kit, MetaGPT, AutoGen, and SOA, under the same incremental software evolution scenario. Each framework evolves a Student–Course system through six controlled changes, repeated 50 times per framework, resulting in 300 executions. This repeated design allows us to analyze not only average behavior, but also variability, outliers, and trade-offs across frameworks.

The benchmark measures execution time, token usage, monetary cost, cached tokens, and API calls. We consider these metrics relevant because agentic development is not only a question of whether an artifact can be generated, but also of how much interaction, computation, and financial cost are required to generate it. In software evolution, this is especially important: if every change requires a costly and slow regeneration process, the framework may become impractical even when its final artifact is acceptable. We also report *sounds good*, a binary structural sanity metric introduced in [11].

Our goal is not only to show whether BAEs are better or worse than other frameworks. Instead, we use BAEs as the main case for studying how different agentic frameworks behave under the same incremental evolution workload. The experiment includes frameworks already compared in the original BAE evaluation, allowing us to observe whether their behavior remains consistent, and also adds MetaGPT, AutoGen, and SOA to broaden the comparison. This design helps us discuss the practical trade-offs among efficiency, structural viability, artifact usability, and stability across repeated executions. In this sense, the paper provides an expanded benchmark for agentic software engineering frameworks, a metric-driven evaluation grounded in prior work, and an empirical analysis of how different orchestration styles behave when software must evolve incrementally rather than be generated only once.

2 Background

2.1 Agentic Software Engineering Frameworks

Agentic software engineering frameworks refer to LLM-based systems that organize software development through agents, roles, tools, specifications, or structured execution workflows. This terminology is close to what previous work calls LLM-based multi-agent systems for software engineering [7]. The systems compared in this paper do not expose the same architecture, but all of them provide an executable framework for using LLMs in software development tasks.

These frameworks differ substantially in their assumptions. ChatDev and MetaGPT are closer to role-based software generation, where agents simulate development roles and exchange messages to produce artifacts [8, 13]. GitHub Spec Kit follows a more specification-driven workflow, where structured requirements guide the generation process [5]. AutoGen provides a more general infrastructure for coordinating LLM-based agents and tool use [15]. SOA represents a different style of agent organization, focused on self-organized collaboration among agents [9]. These differences are important because they affect not only the generated code, but also the number of interactions, the amount of context passed to the model, the output format, and the degree of human involvement expected during execution.

Our comparison does not assume that these frameworks are architecturally equivalent. Instead, we treat them as experimental subjects under the same software evolution workload. This allows us to observe how different orchestration strategies behave when they are asked to produce comparable artifacts under the same prompt sequence, LLM model, and experimental configuration.

2.2 Business Autonomous Entities

BAEs were proposed as an approach for incremental software evolution [6]. A BAE is an LLM-powered agent associated with a business concept, such as *Student*, *Course*, or *Teacher*. Its main distinction is that it maintains a persistent representation of the concept it governs. This representation acts as a semantic anchor for future changes, allowing the BAE to interpret new requirements in relation to what the system already knows, instead of treating each request as an isolated generation task.

The BAE workflow involves human and autonomous roles. The Human Business Expert (HBE) provides requirements in natural language. The BAE interprets the request against its internal representation and decides whether the intent is clear enough to proceed. When confidence is low, the BAE can trigger a clarification loop, asking for additional information before changing the system. This means that BAEs do not remove the human from the process. Instead, they use human input when the autonomous interpretation is uncertain. The Human Software Engineering Expert (HSWE) may also support this process by validating interpretations or architectural assumptions when necessary. Once the change is understood, the BAE decomposes it into implementation tasks and delegates them to Software-Engineering Autonomous Agents (SWEAs), such as database, backend, frontend, or testing agents [6].

For this paper, the relevant aspect of BAEs is not the full architecture, but the evaluation hypothesis behind it. BAEs are designed to reduce unnecessary rebuilding by preserving domain meaning across changes. Their persistent representation, clarification mechanism, SWEA delegation, and semantic alignment checks are intended to support software evolution as a sequence of related changes, rather than as independent prompt-response cycles.

2.3 Benchmarking Agentic Frameworks

Evaluating agentic software engineering frameworks is difficult because different systems report different metrics, run under different assumptions, and sometimes produce artifacts in different formats. Traditional code-generation benchmarks, such as HumanEval [3] or MBPP [1], are useful for measuring programming problem solving, but they do not fully capture the behavior of frameworks that generate multi-file applications, request clarifications, coordinate

agents, or evolve an existing domain model [7, 11]. In particular, incremental software evolution requires observing whether the framework can incorporate new requirements while preserving previous design decisions and producing usable artifacts.

Our evaluation perspective follows the metric-oriented view introduced in [11], which organizes metrics for LLM-based multi-agent frameworks into *Outcome*, *Process*, *Product*, and *Framework* dimensions. In this paper, we instantiate this perspective through a controlled incremental-evolution benchmark. Rather than replacing existing benchmarks, our goal is to place representative frameworks under the same workload and observe practical trade-offs across efficiency, interaction cost, structural viability, and artifact usability.

3 Study Design

3.1 Benchmark Task

The benchmark evaluates how different agentic software engineering frameworks behave in an incremental software evolution scenario. The task is based on a simple Student–Course domain implemented as a Python web application using FastAPI and SQLite. Instead of asking each framework to generate a complete system in a single prompt, the benchmark is organized as a sequence of six sprints, where each sprint introduces a new requirement that must build on the previous ones.

The six sprints are: (i) create a Student entity with a required name field; (ii) add a required email field to Student; (iii) create a Course entity with name and level fields; (iv) add a relationship between Student and Course; (v) create a Teacher entity with name and email fields; and (vi) add a relationship between Course and Teacher.

The benchmark targets Python 3.12, FastAPI, and SQLite, requires JSON responses, automatic database schema creation, and preservation of existing data when the schema evolves. These requirements were intentionally kept simple, but they force the frameworks to deal with changes that are typical in software evolution, such as adding fields, creating new entities, updating relationships, and preserving previous data.

3.2 Execution and Measurement Protocol

We compare BAEs with five agentic software engineering frameworks: ChatDev, GitHub Spec Kit, MetaGPT, AutoGen, and SOA. The BAE approach has been previously described and evaluated in [6]; here, we use the same approach as an experimental subject and apply the benchmark protocol described above consistently across all frameworks. Each framework is executed 50 times, and each run corresponds to a full execution of the six-sprint evolution sequence. This results in 300 runs in total. The repeated design is used to reduce sensitivity to isolated outliers and to observe run-to-run variability, since LLM-based systems may behave differently even under the same prompt sequence and configuration.

All executions use the same prompt sequence, LLM model, and experimental configuration whenever supported by the framework. The benchmark uses GPT-5-mini as its LLM, and the Python runtime is Python 3.12. We selected this model because it was compatible with the adapted frameworks and allowed the full benchmark to be executed at a manageable cost. To reduce configuration differences, we forked the evaluated frameworks, except GitHub Spec Kit, and adapted their model interface when necessary so that they could run with the same OpenAI model configuration.

Some frameworks may request clarification during execution, while others continue by making their best guess. To avoid introducing variation from live human responses, we use a standardized clarification protocol. When a framework requests additional information, it receives the same expanded specification, which instructs it to follow the prompt requirements, use conventional implementation choices for unspecified details, preserve data across schema changes, prefer simple working solutions, and apply reasonable professional judgment when specifications are ambiguous. This keeps clarification behavior controlled across runs.

We collect *Process* and *Outcome* metrics [11] from each execution. The benchmark considers six metrics: **Execution Time**, **Token Usage**, **Monetary Cost**, **API Calls**, **Cached Tokens**, and ***Sounds Good***. Together, these metrics capture complementary aspects of framework behavior, including computational efficiency, resource consumption, interaction overhead, and minimum artifact viability. This allows the benchmark to compare frameworks not only by whether they produce an acceptable artifact, but also by the resources and interactions required to achieve it.

Execution time is measured as wall-clock time for the full six-sprint run. Token usage is recorded as input, output, and total tokens. Monetary cost is computed from token usage according to the provider pricing model used during the experiment. API calls count the number of LLM requests made by the framework, while cached tokens are recorded when reported by the provider. We also compute *Sounds Good*, which checks whether the generated artifact satisfies a minimal set of prompt requirements through static inspection, including the presence of a database artifact, Python source files, FastAPI usage, a minimum number of generated files, and the absence of errors in relevant logs. This metric does not measure semantic correctness, but provides a practical minimum-viability signal: when *Sounds Good* is false, the generated artifact has failed at least one explicit structural requirement and is unlikely to be immediately useful without manual intervention. The check is performed by a static scanner after the framework has completed its run and all associated metrics have been tallied.

Additional experimental material, including data and visualizations beyond those reported in this paper, will be made available to support open science and allow future work to inspect, reuse, or extend the benchmark.

4 Results and Discussion

4.1 Structural Viability

As mentioned previously, *sounds good* is a reasonable indicator of structural viability and an initial evaluation metric. According to our experiment, as seen in Figure 1, ChatDev achieved the highest structural viability, successfully satisfying the structural requirements in all executions. BAEs and AutoGen followed closely, differing by only a single unsuccessful run, while GitHub Spec Kit also produced structurally acceptable artifacts in most executions despite a lower success rate, at **44 out of 50**. In contrast, SOA and MetaGPT exhibited substantially lower structural viability under the evaluated configuration, with the former scoring **22 out of 50**, and the latter scoring **0 out of 50**. It is important to note that, for

MetaGPT, this score is a direct result of its output format, which is different from the other frameworks, as it does not generate Python source files directly. Instead, it produces code embedded in log-style outputs, requiring the user to manually extract, copy, and paste artifacts before they can be executed or inspected. Therefore, it still exhibits lower performance in this metric.

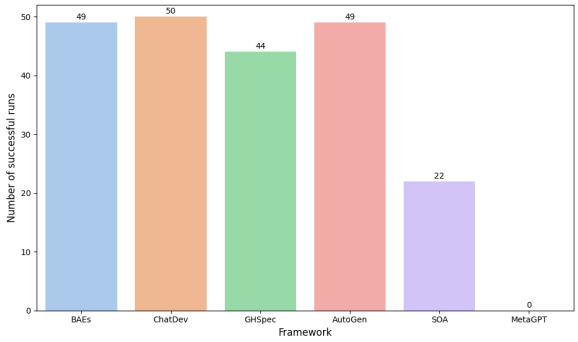

Figure 1: *Sounds Good* result over 50 runs for all frameworks

Another significant observation is that the gap between ChatDev, BAEs, and AutoGen is relatively small when considering only structural viability. All three frameworks consistently generated artifacts that satisfied the benchmark requirements in nearly every execution. This suggests that, for this incremental evolution task, multiple orchestration strategies are capable of producing structurally acceptable outputs. Therefore, structural viability alone is insufficient for distinguishing practical performance, motivating the analysis of efficiency and resource consumption presented in the following subsection.

4.2 Efficiency and Resource Consumption

While structural viability indicates whether a framework produces an artifact that satisfies the benchmark requirements, practical viability also depends on the resources required to achieve that result. Table 1 summarizes the descriptive statistics for all evaluated metrics over the 50 executions of each framework. The results already indicate substantial differences in resource consumption and execution behavior, while the accompanying standard deviations provide a first indication of execution stability. The following figures examine these differences in greater detail through distribution and efficiency analyses.

The box plot in Figure 2 shows that ChatDev and GitHub Spec Kit consistently occupy the higher end of the cost distribution, indicating that their orchestration strategies require substantially greater resource consumption across repeated executions. BAEs and AutoGen, on the other hand, achieve considerably lower execution costs while maintaining structural viability comparable to ChatDev. Although AutoGen occasionally reaches slightly lower costs, BAEs exhibit a noticeably tighter distribution, indicating more stable behavior across runs. SOA and MetaGPT require fewer resources than ChatDev but still display broader variability or lower structural success, reducing the practical benefit of their lower average costs.

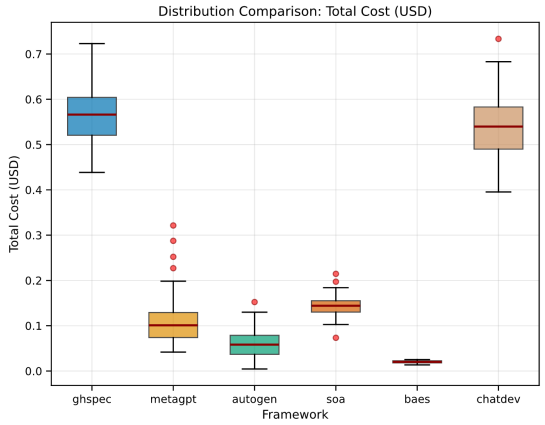

Figure 2: Total cost comparison across frameworks

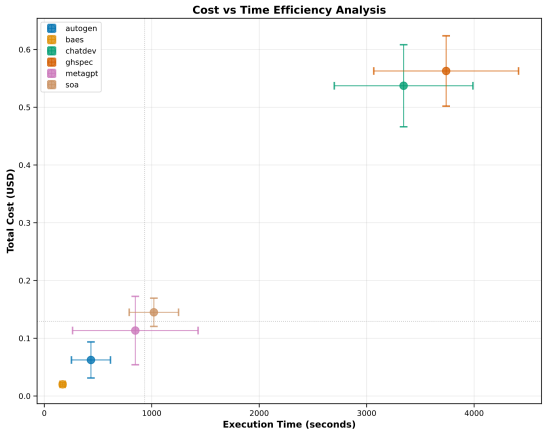

Figure 3: Cost vs Time Efficiency across all frameworks

The efficiency plot in Figure 3 reinforces these observations by jointly considering execution time and monetary cost. ChatDev provides the highest structural viability but reaches this result through longer executions and higher resource consumption. Compared with the other evaluated frameworks, BAEs and AutoGen occupy a more favorable region of the efficiency space, combining relatively short execution times with low execution costs while maintaining nearly identical *sounds good* rates. This illustrates that frameworks producing similar structural outcomes may require substantially different computational effort.

Taken together, these results indicate that evaluating agentic software engineering frameworks solely by artifact viability may hide important differences in practical efficiency. Frameworks that produce nearly identical structural outcomes can differ considerably in execution time, interaction volume, and monetary cost.

4.3 Trade-offs Across Frameworks

Figure 4 summarizes the benchmark through a normalized radar chart that combines the evaluated metrics into a single comparative view. Rather than identifying a single best framework, the

Table 1: Per-framework summary statistics over n=50 runs each. Values are reported as Mean ± SD.

Metric	BAEs	ChatDev	GHSpec	AutoGen	SOA	MetaGPT
Execution Time (s)	170.57 ± 24.31	3343.86 ± 644.96	3739.77 ± 673.70	434.19 ± 181.61	1019.79 ± 229.47	847.04 ± 584.10
API Cost (USD)	0.02021 ± 0.00319	0.53723 ± 0.07104	0.56279 ± 0.06091	0.06239 ± 0.03125	0.14495 ± 0.02445	0.22337 ± 0.05920
API Calls	5.92 ± 0.94	77.10 ± 5.42	94.52 ± 9.62	20.68 ± 10.35	37.76 ± 5.98	33.30 ± 11.74
Tokens In	26228.18 ± 2435.46	374307.52 ± 77354.63	340510.06 ± 32509.05	87060.12 ± 79420.21	54578.48 ± 8763.51	166537.08 ± 130920.61
Tokens Out	8957.60 ± 1426.50	224816.52 ± 12537.39	256003.62 ± 28407.48	26197.46 ± 12537.39	68247.90 ± 11541.34	36309.82 ± 28407.48
Total Tokens	35185.78 ± 3812.41	599124.04 ± 103290.61	596513.68 ± 59230.32	113257.58 ± 91390.65	122826.38 ± 20023.31	202846.90 ± 143811.99
Cached Tokens	7848.96 ± 2938.76	26565.12 ± 18682.90	152622.08 ± 18164.83	52298.24 ± 62702.25	23080.96 ± 4978.18	3937.28 ± 7397.27

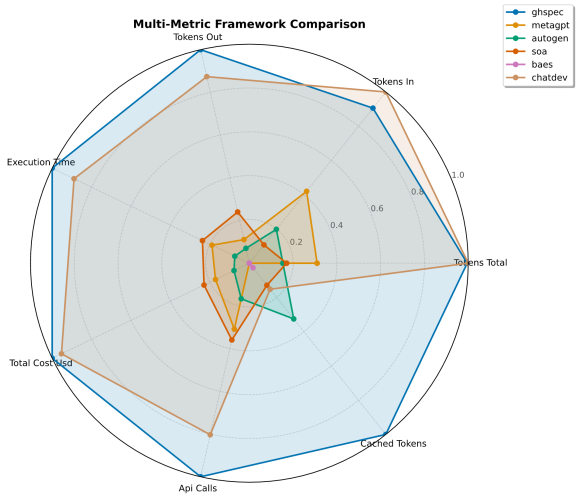

Figure 4: Normalized Resource Consumption Across Frameworks

radar chart highlights distinct performance profiles resulting from different orchestration strategies.

ChatDev stands out for its excellent structural viability, but this performance is accompanied by comparatively high execution time, token consumption, API interactions, and monetary cost. BAEs and AutoGen occupy a more resource-efficient region of the chart, achieving similar structural viability while requiring substantially fewer computational resources. GitHub Spec Kit remains structurally viable in most runs, but its profile is characterized by high token usage and execution cost, with the former being significantly higher than other frameworks, considering cached tokens. SOA and MetaGPT exhibit lower structural viability under the evaluated configuration, suggesting that their orchestration strategies are less effective for this particular incremental evolution workload.

Overall, the benchmark reinforces that no framework dominates every evaluated dimension. Instead, each framework reflects different design priorities, balancing structural robustness, computational efficiency, interaction complexity, and execution stability in different ways. From this perspective, the main contribution of the benchmark is not to establish an absolute ranking, but to reveal the trade-offs involved in agentic software engineering. Considering structural viability together with cost, execution time, token consumption, API interactions, and stability provides a more complete understanding of framework behavior than any individual metric in isolation.

5 Threats to Validity

Although this benchmark follows a controlled and repeated experimental design, some limitations should be acknowledged. The first limitation concerns the benchmark task. We evaluate the frameworks in a single incremental software evolution scenario, centered on a Student–Course domain. This choice makes the comparison controlled and reproducible, but it also limits external validity. Different domains, larger systems, richer business rules, or other types of evolution tasks could lead to different framework behavior. For this reason, the results should be understood as evidence for this class of incremental evolution task, not as a universal ranking of agentic software engineering frameworks.

A second limitation concerns the interaction between prompts, clarification, and the LLM model. LLM-based systems are sensitive to how task requirements and intermediate context are formulated and presented to the model [14]. In our benchmark, we use a fixed prompt sequence and a standardized clarification protocol to keep the comparison fair across frameworks. This reduces human variation, but it also means that other prompt formulations or clarification strategies could affect the results. Similarly, all executions use GPT-5-mini. We do not treat this as a flaw in the study, because the goal is to compare framework behavior under the same engine, not to benchmark different models. Still, different models or providers may change absolute costs, token usage, execution time, or outcome quality. Future work should investigate whether the same trade-offs remain stable across other models and prompt designs.

Finally, there are limitations related to the selected metrics. We measure execution time, token usage, monetary cost, API calls, cached tokens, and *sounds good*. These metrics capture efficiency, interaction cost, and minimum structural viability, but they do not fully capture semantic correctness, maintainability, security, or long-term evolvability. In particular, *sounds good* is intentionally used as a binary structural sanity check, not as a substitute for deeper functional or human evaluation. Therefore, passing *sounds good* means that an artifact satisfies the minimum structural requirements of the benchmark, not that it is semantically complete or production-ready. Expanding the metric set with richer semantic,

functional, and human-centered measures is a natural direction for future work.

6 Conclusion and Future Work

This paper presented an extended benchmark of BAEs against five representative agentic software engineering frameworks: ChatDev, GitHub Spec Kit, MetaGPT, AutoGen, and SOA. The goal was to evaluate how BAEs and other frameworks behave under the same incremental software evolution workload. Across 300 executions, the benchmark allowed us to observe not only average efficiency, but also structural viability, interaction cost, and run-to-run stability.

The results show that no framework dominates all dimensions. ChatDev achieves the highest *sounds good* rate, but this comes with higher execution time and monetary cost. BAEs and AutoGen achieve nearly the same *sounds good* rate while remaining more efficient in *Process* metrics such as time and cost. GitHub Spec Kit remains structurally viable in most executions, but its cost profile is less favorable. SOA and MetaGPT show lower structural viability in this setup. These results suggest that evaluating agentic frameworks only by whether they can generate an artifact is not enough. A useful comparison must also consider how expensive, slow, interaction-heavy, stable, and immediately usable the generated artifacts are. Other evaluation axes, such as semantic correctness, maintainability, security, and human judgment, are also relevant and could complement this benchmark.

From the BAE perspective, the results are encouraging but should not be interpreted as a claim of superiority. BAEs are competitive with strong and widely used frameworks, including frameworks backed by large research or industrial groups. This is relevant because BAEs were designed for a specific purpose, incremental software evolution and ontology-aware comprehension, rather than general code generation alone. In our benchmark, this design leads to a balanced profile, in which BAEs remain structurally competitive while showing favorable efficiency and stability. However, this conclusion is bounded by the task, metrics, model, and protocol used in this study, and different scenarios could lead to different results.

Future work should extend this evaluation in several directions. The benchmark should be replicated with other domains and richer evolution scenarios, including systems with more complex business rules and longer dependency chains. Different prompt designs and clarification strategies should also be tested, since LLM-based systems are sensitive to how requirements are expressed and refined. Although this paper intentionally fixes the LLM model to isolate framework behavior, future studies should investigate whether the observed trade-offs remain stable across other models and providers. Finally, the metric set should be expanded with deeper semantic, functional, maintainability, and human-centered evaluations to capture aspects that structural metrics cannot capture.

Overall, this paper provides evidence that controlled, repeated, and metric-driven benchmarks are necessary to understand agentic software engineering frameworks. The main result is not that one framework is always better than the others, but that different orchestration strategies lead to different trade-offs. In incremental software evolution, frameworks should not only generate code, but also preserve previous decisions, control cost and interaction, and produce artifacts suitable for continued evolution.

Artifact Availability

The artifacts are available as a replication package in a public repository: https://zenodo.org/records/21876580.

Acknowledgements

This work was supported by the State University of Ceará. We used LLMs for language refinement, but all analysis and interpretation remain our sole responsibility.

References

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[2] Frederick P. Brooks. 1987. No Silver Bullet Essence and Accidents of Software Engineering. *Computer* 20, 4 (1987), 10–19. doi:10.1109/MC.1987.1663532

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Hodas, Sandhini Chen, Alec Radford, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[4] 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)*. IEEE, 31–53.

[5] GitHub. 2026. *Spec Kit: A toolkit for Spec-Driven Development.* https://github.com/github/spec-kit

[6] Anderson Martins Gomes, Paulo Henrique M. Maia, Vitor Fontenele de Oliveira Linhares, and Ingrid Lima Vieira. 2026. From Stateless Code Generation to Living Ontologies: Business Autonomous Entities in Action. In *Proceedings of the 2026 International Workshop on Agentic Engineering (AGENT '26)*. Association for Computing Machinery, New York, NY, USA, 125–134. doi:10.1145/3786167.3788408

[7] Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–30. doi:10.1145/3712003

[8] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. 2023. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

[9] Yoichi Ishibashi and Yoshimasa Nishimura. 2024. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization. *arXiv preprint arXiv:2404.02183* (2024).

[10] M. M. Lehman. 1996. Laws of Software Evolution Revisited. In *Proceedings of the 5th European Workshop on Software Process Technology (EWSPT '96)*. Springer-Verlag, Berlin, Heidelberg, 108–124.

[11] Ingrid Lima, Vitor Linhares, Anderson Martins Gomes, and Paulo Henrique Maia. 2026. A Catalogue of Evaluation Metrics for LLM-Based Multi-Agent Frameworks in Software Engineering. In *Proceedings of the 2026 International Workshop on Agentic Engineering (AGENT '26)*. Association for Computing Machinery, New York, NY, USA, 120–124. doi:10.1145/3786167.3788430

[12] Mahmoud Mohammadi, Yipeng Li, Jane Lo, and Wendy Yip. 2025. Evaluation and Benchmarking of LLM Agents: A Survey. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2* (Toronto ON, Canada) *(KDD '25)*. Association for Computing Machinery, New York, NY, USA, 6129–6139. doi:10.1145/3711896.3736570

[13] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924* (2023).

[14] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2024. Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design or: How I learned to start worrying about prompt formatting. In *International Conference on Learning Representations*, Vol. 2024. 25055–25083.

[15] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.