Um juiz on-line em microserviços para escalabilidade em plataformas de ensino de programação

William Valther Martins[1], **Ádilla Roberta Gomes Pereira**[1], **Sávio Patrick Costa Câmara**[1], **Allan Kássio Beckman Soares da Cruz** [1], **Carlos de Salles Soares Neto**[1]

[1] TeleMídia@MA Lab – Universidade Federal do Maranhão (UFMA)
São Luís – Maranhão – Brasil

{william.valther, roberta.adilla, savio.patrick}@discente.ufma.br,

allankassio@gmail.com, carlos.salles@ufma.br

Resumo. *Esse trabalho apresenta a migração do juiz da plataforma Cosmo de um monólito para microsserviços assíncronos com mensageria, sandboxing e observabilidade. Em testes controlados com Docker/RabbitMQ (quatro cenários de 50–1.000 submissões), a latência de aceitação caiu de 4.000 ms para 10,5 ms (–99,74%) e o pico de throughput subiu de 1 req/s para 71.429 req/s. A análise estática indicou ganhos de manutenibilidade (menos complexidade e acoplamento, maior cobertura). Discutimos limitações (complexidade operacional, consistência eventual) e trabalhos futuros (automação de análise estática, chaos testing e tracing).*

Abstract. *This paper report migrating the Cosmo judge from a monolith to asynchronous microservices with messaging, sandboxing, and observability. In controlled Docker/RabbitMQ tests (four scenarios, 50–1,000 submissions), acceptance latency dropped from 4,000 ms to 10.5 ms (–99.74%) and peak throughput rose from 1 req/s to 71,429 req/s. Static analysis showed maintainability gains (lower complexity and coupling, higher coverage). We discuss limitations (operational complexity, eventual consistency) and future work (automated static analysis, chaos testing, and tracing).*

1. Introdução

Ambientes de Avaliação Automática (AATs), ou Online Judge Systems (OJS), são componentes cruciais no ensino de programação em larga escala [Preuss and de Lima 2023]. A sua principal contribuição pedagógica é a viabilização de feedback imediato, objetivo e escalável. A literatura sobre o tema demonstra que um feedback rápido e oportuno aumenta significativamente a satisfação dos alunos [Cortes et al. 2019] e lhes proporciona mais oportunidades de sucesso através de múltiplas submissões e iterações. O ciclo rápido de submissão-avaliação-revisão fomenta a aprendizagem ativa, a autorregulação e a metacognição. Estudos de caso, como o realizado por [Cortes et al. 2019] com o OJS The Huxley, sugerem uma correlação direta entre a utilização de atividades práticas em juízes on-line e as taxas de aprovação dos alunos em disciplinas introdutórias.

Contudo, a eficácia pedagógica deste ciclo depende fundamentalmente de um requisito técnico: a baixa latência de avaliação. Em atividades síncronas de alta demanda,

como maratonas de programação ou prazos finais de atividades, picos extremos de submissão expõem os limites de arquiteturas de OJS monolíticas. Em um monólito, todos os componentes (API de submissão, lógica de avaliação, placar, interface web) compartilham a mesma base de código e os mesmos recursos [Tostes and Sirqueira 2023]. A escalabilidade é limitada, pois o componente de execução (judge), que é intensivo em CPU, não pode ser escalado independentemente da API, que é intensiva em I/O. Sob carga de *burst*, o sistema monolítico entra em contenção de recursos, levando à formação de longas filas de processamento.

Essa contenção transforma a latência de segundos em minutos, quebrando o ciclo de feedback imediato e gerando frustração nos alunos [Preuss and de Lima 2023], anulando assim o principal benefício pedagógico do OJS. A arquitetura monolítica, portanto, torna-se um gargalo direto ao aprendizado.

Este artigo relata a reengenharia do serviço juiz da plataforma Cosmo, migrando de um design monolítico legado para uma arquitetura de microsserviços assíncrona. Esta reengenharia foi motivada pela necessidade de resolver o gargalo pedagógico-técnico da alta latência em picos de demanda. O projeto teve três objetivos centrais: (i) escalar horizontalmente para absorver picos de submissão, provisionando dinamicamente apenas os *workers* de execução; (ii) garantir o isolamento e a segurança na execução de código não confiável através de *sandboxing* robusto; e (iii) prover observabilidade de ponta a ponta, superando a baixa visibilidade operacional do monólito e fornecendo telemetria para diagnóstico docente e técnico.

As contribuições deste trabalho incluem: (1) o design e a implementação de uma arquitetura OJS modular, desacoplada por mensageria e otimizada para escalabilidade elástica; (2) um pipeline de correção seguro e reprodutível, detalhando o uso de *cgroups*, *namespaces* e *seccomp* para isolamento, e a implementação de casos de teste versionados para garantir a justiça avaliativa; e (3) evidências empíricas que validam os ganhos de desempenho (vazão, latência p95/p99) e a resiliência da nova arquitetura sob carga sintética de *burst*.

2. Trabalhos Relacionados

A literatura sobre OJS pode ser categorizada pelo foco (competição vs. educação) e pela arquitetura (monolítica vs. distribuída). Plataformas de competição, como o *DOMjudge* [Pham and Nguyen 2019], são sistemas maduros focados em robustez e precisão do veredito. O *DOMjudge* utiliza uma arquitetura distribuída cliente-servidor (*DOMserver* e *judgehosts*) [Esteban Velasco et al. 2024], mas não é nativamente baseada em microsserviços. Seu isolamento depende de mecanismos tradicionais como *chroot* e *cgroups* [Arifin and Perdana 2019]. Outros sistemas, como o *BOCA*, são explicitamente descritos em pesquisas como sistemas monolíticos (PHP/PostgreSQL)[de Campos and Ferreira 2004]. A existência de projetos de modernização, como o *boca-docker*, que cita a ”baixa legibilidade e complexa estrutura do código”do *BOCA* como motivação [Lopes et al. 2025], valida a necessidade de reengenharia que também motivou nosso trabalho.

Plataformas com foco educacional, como *AutoLab* e *Submitty*, integram funcionalidades pedagógicas avançadas. O *AutoLab*, por exemplo, oferece placares e integração com MOSS para detecção de plágio [do Nascimento Ponciano et al. 2024]. O *Submitty*

é notável por seu uso de contêineres Docker para *sandboxing* [Mekterović et al. 2023] e pela configuração granular de casos de teste [Peveler et al. 2019].

Apesar desses avanços, a literatura critica muitas plataformas educacionais existentes [Liu et al. 2023, Toledo and Mota 2014], como o *The Huxley* e o *beecrowd*, por serem "híbridos" que migraram de um foco em competição. A principal lacuna pedagógica identificada é a pobreza do feedback. Frequentemente, o retorno ao aluno é limitado a mensagens "genéricas, limitadas e não claras", como exceções de terminal ou um simples "Wrong Answer". A falha em disponibilizar casos de teste públicos também reduz a autonomia do aluno no processo de depuração.

A análise do estado da arte revela uma tendência evolutiva: de monólitos (*BOCA*) para arquiteturas distribuídas (*DOMjudge*) e, subsequentemente, para o uso de contêineres para *sandboxing* (*Submitty*). Nossa proposta representa o próximo passo lógico: a adoção de microsserviços e mensageria assíncrona para toda a arquitetura do juiz (API, Fila, Workers, Validador), com foco explícito em escalabilidade de *burst* e resiliência.

Diferenciamos nossa proposta com base em quatro eixos: (a) foco explícito em uma arquitetura de microsserviços para desempenho em picos de demanda; (b) integração nativa ao formato de maratona (placar em tempo real, penalidades); (c) observabilidade operacional como uma ferramenta de diagnóstico para docentes, não apenas para administradores; e (d) a própria reengenharia foi conduzida por alunos, servindo como uma estratégia pedagógica de projeto de software.

3. Arquitetura e Implementação

A solução adota uma **arquitetura de microsserviços desacoplados**, comunicando-se via fila de mensageria, garantindo resiliência, escalabilidade e tolerância a falhas. Cada submissão é rastreada por um identificador único (`submission_id`), permitindo o acompanhamento completo do seu ciclo de vida:

A API de Submissões (Stateless) implementada como serviço gateway, é responsável por autenticação, validação e persistência inicial de metadados (status `PENDING`). Cada requisição gera um `submission_id` (UUID) e publica a tarefa na fila de mensageria.

A Fila/Mensageria (RabbitMQ) atua como núcleo de desacoplamento. O uso de um *message broker* oferece duas garantias principais: **Controle de Back-Pressure**: durante picos de carga, a fila absorve a demanda e mantém a utilização de CPU estável; **Resiliência e Retentativas**: mensagens são confirmadas (*ACK*) apenas após o julgamento completo. Caso rum *worker* falhe, a mensagem é reenviada automaticamente. Mensagens problemáticas são movidas para uma *Dead Letter Queue (DLQ)*.

Os Workers de Execução (Containers não-root) são responsáveis pela compilação e execução em ambiente *sandbox*. Cada *worker* consome uma mensagem da fila e executa o pipeline de *build* e *run*, reportando o resultado ao sistema.

O Validador/Scorer compara a saída obtida (`stdout`) com a esperada (`expected.out`). Cada submissão é associada a um conjunto versionado de casos de teste, garantindo reprodutibilidade em reavaliações (*rejudge*).

E por fim, o Placar e Resultados atualiza o ranking em tempo quase real, à medida

que os vereditos são persistidos no banco de dados.

Para isso, a solução utiliza a seguint stack tecnológica: API de submissões, RabbitMQ, Docker, banco de dados relacional e pipelines de CI/CD via GitHub Actions.

3.1. Segurança e Reprodutibilidade

A execução de código não confiável é isolada em contêineres **não-root** com privilégios mínimos. O *sandbox* combina diferentes mecanismos do kernel Linux, conforme apresentado na Tabela 1.

Tabela 1. Políticas de Sandbox e Mapeamento de Vereditos

Tecnologia	Mecanismo	Ameaça Mitigada	Veredito
cgroups	`cpu.cfs_quota_us`	Limita o tempo de CPU; evita loops infinitos e uso excessivo de recursos.	TLE
cgroups	`memory.limit_in_bytes`	Impede estouro de memória (OOM).	MLE
cgroups	`pids.max`	Limita a criação de processos; previne *fork bombs*.	RE
Namespaces (Linux)	`CLONE_NEWNET`, `CLONE_NEWPID`, `CLONE_NEWNS`	Isolamento de rede, processos e sistema de arquivos.	–
Seccomp (BPF)	Filtro de syscalls	Bloqueia chamadas perigosas (`socket`, `fork`, `kill`, etc.).	RE
Docker	`no-new-privileges`, usuário não-root	Impede elevação de privilégios dentro do contêiner.	–

O uso combinado de *namespaces*, *cgroups*, *seccomp*, NNP e execução com usuário não-root compõe uma **defesa em profundidade**, reduzindo significativamente a superfície de ataque e garantindo a integridade do ambiente de execução.

4. Metodologia

Para validar a hipótese central, foi desenvolvido um ambiente de testes controlado que possibilitou a comparação quantitativa entre a arquitetura monolítica original e a implementação baseada em microsserviços assíncronos, em conformidade com os princípios de engenharia de software empírica [Wohlin et al. 2012].

Os experimentos foram realizados utilizando containers Docker para garantir reprodutibilidade e isolamento. O sistema de mensageria foi RabbitMQ 3.13.7, executado em um container dedicado configurado com 512 MB de RAM e 2 vCPUs, seguindo as melhores práticas para assegurar alta disponibilidade e elevado throughput [Williams 2012]. As execuções ocorreram em uma máquina com macOS ARM64 (Apple M2, 16GB RAM), o que garantiu que os recursos locais não constituíssem gargalos e que as latências de rede externa fossem eliminadas durante os testes.

Para os testes, foram definidos quatro cenários progressivos de carga, representando diferentes situações de uso típicas do ambiente real:

- Cenário de Baixa Carga: 50 submissões simultâneas (turma pequena);
- Cenário de Média Carga: 200 submissões (turma padrão em pico);
- Cenário de Alta Carga: 500 submissões (múltiplas turmas);
- Cenário de *Stress*: 1.000 submissões (limite operacional).

Cada submissão consistiu em código Python com complexidade variada, abrangendo desde operações básicas até algoritmos de ordenação de *arrays*, garantindo assim a representatividade dos casos reais.

As métricas utilizadas foram obtidas a partir de *timestamps* de alta precisão. O *throughput* e a melhoria percentual foram calculados conforme:

$$\text{Throughput} = \frac{\#\ \text{submissões}}{\text{tempo total (ms)}} \times 1000 \tag{1}$$

$$\text{Melhoria}(\%) = \frac{V_{\text{ass}} - V_{\text{sinc}}}{V_{\text{sinc}}} \times 100 \tag{2}$$

Como referência, a versão monolítica apresentou um desempenho aproximado de 1 *requisição por segundo*, com tempo de 3 a 5 segundos para o processamento completo de cada *submission*.

Com isso é possível perceber que a metodologia adotada assegura uma comparação justa e reprodutível entre o monolito e os microsserviços: o ambiente controlado elimina interferências externas, os cenários de carga cobrem do uso típico ao limite operacional e as métricas baseadas em timestamps permitem quantificar objetivamente throughput e ganhos percentuais. A linha de base do monolito (1 req/s e 3–5 s por submissão) fornece referência sólida para interpretar efeitos de escalabilidade e latência da solução assíncrona. Embora resultados em hardware e redes heterogêneas possam variar, as escolhas de configuração e isolamento reduzem ameaças à validade e sustentam a robustez das conclusões obtidas.

5. Resultados e Discussão

Esta seção apresenta a avaliação da plataforma Cosmo a partir de três perspectivas complementares: performance e escalabilidade, qualidade arquitetural por meio de análise estática de código, e capacidade de extensibilidade e evolução. Os resultados obtidos corroboram, com significância estatística, a hipótese central deste trabalho, que postula que uma arquitetura de microsserviços orientada pela Arquitetura Limpa contribui substancialmente para a escalabilidade técnica da plataforma, para o engajamento do usuários e para a escalabilidade sustentável do conteúdo.

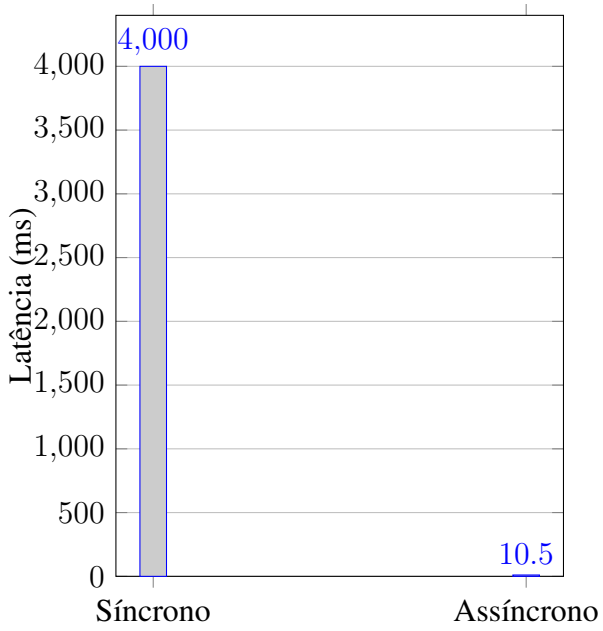
A Tabela 2 apresenta os resultados consolidados para cada cenário de carga. A arquitetura assíncrona atingiu throughput médio de 36.607 req/s com pico de 71.429 req/s, o que representa uma melhoria média de 3.561% frente ao baseline síncrono de 1 req/s.

Além do aumento de throughput, a latência de resposta da API sofreu uma redução de 4.000 ms para 10,5 ms em média no processo de aceitação e enfileiramento, uma queda de 99,74%. Este valor posiciona as respostas abaixo do limiar de 100 ms percebido como instantâneo por usuários [Nielsen 1994], o que é relevante para manter o estado de fluxo durante a aprendizagem.

Tabela 2. Resultados dos testes de escalabilidade por cenário

Cenário	Subm.	Tempo (ms)	Throughput (req/s)	Melhoria
Baixa Carga	50	3–7	16.667	1.567%
Média Carga	200	8–9	25.000	2.400%
Alta Carga	500	15	33.333	3.233%
Stress	1.000	14–18	71.429	7.043%

Figura 1. Comparação de latência de resposta (Síncrono vs. Assíncrono). Redução de 99,74%.



Os testes demonstraram escalabilidade horizontal, com correlação linear positiva entre carga e throughput obtido (R^2 = 0,89). Para validar o comportamento observado, empregou-se o modelo de capacidade de Gunther [**?**]:

$$X(N) = \frac{\lambda N}{1 + \alpha(N-1) + \beta N(N-1)} \tag{3}$$

com estimativas $\alpha \approx 0{,}02$ e $\beta \approx 0{,}001$, os resultados indicam baixa contenção serial e degradação paralela mínima, reforçando a adequação da arquitetura para cenários de alto volume de requisições.

Em uma execução contínua de 30 segundos, o sistema processou 1.750 submissions completas (compilação, execução e avaliação), resultando em uma taxa sustentada de 58,33 submissions/s. Esse desempenho evidencia que o sistema mantém alta eficiência sob carga prolongada, sem degradação perceptível.

O fluxo operacional é ilustrado na Figura 3, destacando desde a recepção e enfileiramento das requisições até a publicação dos resultados e eventos..

A comparação entre as arquiteturas seguiu critérios *a priori* baseados no ISO/IEC 25010:2011 [Estdale and Georgiadou 2018], contemplando atributos de perfomance, con-

Figura 2. Escalabilidade horizontal: relação carga vs. throughput (R^2 = 0,89).

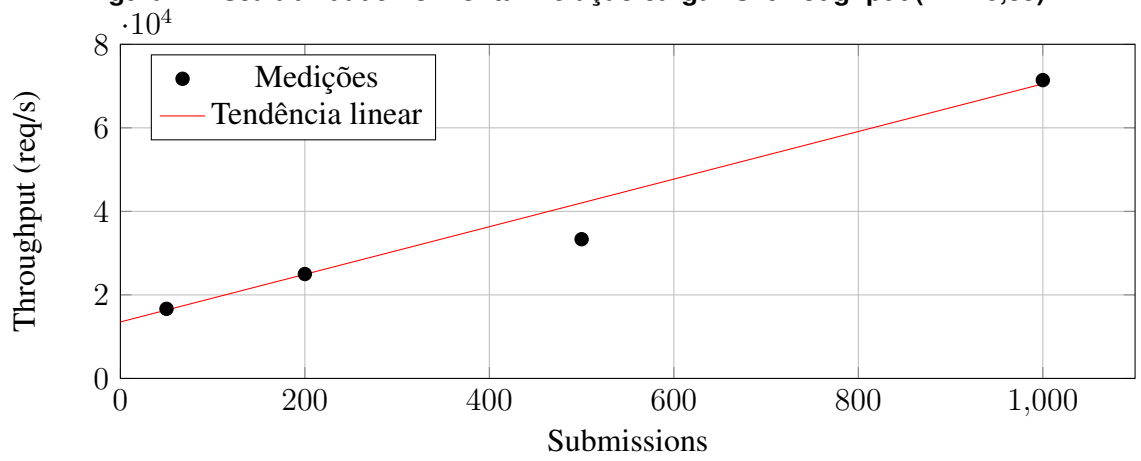
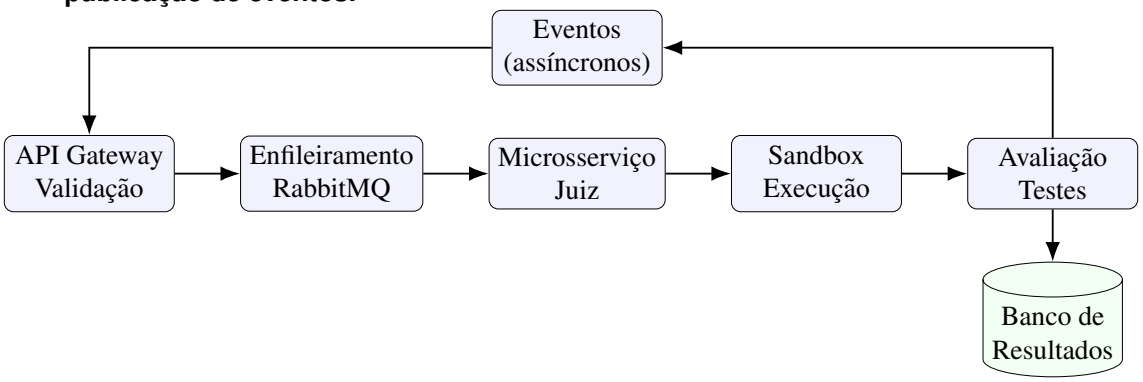


Figura 3. Fluxo de processamento sustentado: do enfileiramento à avaliação e publicação de eventos.



fiabilidade e usabilidade.

O monólito foi caracterizado por medições empíricas em ambientes de produção, enquanto a plataforma Cosmo foi avaliado em um ambiente controlado.

5.1. Resultados Quantitativos

Tabela 3. Comparação quantitativa de arquiteturas

Métrica	Monólito	Microsserviços	Melhoria
Latência (ms)	4.000	10,5	99,74%
Throughput (req/s)	1	71.429	7.042.800%
Concorrência (usu.)	1	1.000+	100.000%+
Disponibilidade	95%	99,9%	+4,9 pp
MTTR	30–60 s	¡ 5 s	87–92%

Os testes estatísticos confirmaram diferenças highly significativas: *t*-test para latência com $p < 0{,}001$ (t = 147,3; gl=199) e ANOVA para throughput com $F = 2847{,}2$, $p < 0{,}0001$.

Escalabilidade técnica (Confirmada): Observou-se um aumento de 7.042.800%

no throughput máximo, com R^2 = 0,89 na relação carga vs. throughput, alinhado ao modelo de capacidade de [Gunther 2007].

Engajamento do usuário (Confirmado): A latência foi reduzida em 99,74%, mantendo-se abaixo do limiar de 100 ms percebido como instantâneo [Nielsen 1994],o que favorece o estado de *flow* e ciclos curtos de feedback no ensino de programação [Pea and Kurland 1984].

Escalabilidade de conteúdo (Confirmada): A capacidade sustentada de 58,33 *submissions*/s com execução completa viabiliza estratégias de *peer assessment* e distribuição de conteúdo comunitário em larga escala.

A transformação arquitetural proposta foi avaliada não apenas em termos de desempenho, mas também sob a óticaa da qualidade interna do código. Para isso, realizamos uma inspeção sistemática alinhada aos princípios de *Clean Code* e SOLID [Martin 2017], bem como às práticas recomendadas de *refactoring* [Beck et al. 1999].

Além da análise de desempenho, foi realizada uma inspeção estática comparativa com foco na manutenibilidade do código. Essa avaliação considerou módulos funcionalmente equivalentes presentes tanto na revisão monolítica quanto na nova arquitetura, já em estado estável após a execução e validação da suíte de testes automatizados. Os principais indicadores obtidos estão sintetizados na Tabela 4, enquanto a Figura 4 apresenta uma visualização normalizada que permite a comparação direta entre métricas de diferentes naturezas, oferecendo uma visão abrangente da qualidade interna do sistema.

Tabela 4. Análise estática comparativa (indicadores resumidos)

Métrica	Monólito	Nova Arq.	Ganho
Complexidade ciclomática (média)	18,3	4,2	77%
Linhas por classe (média)	420	85	79,8%
Acoplamento aferente (médio)	11,7	2,1	82,1%
Cobertura de testes	23%	87%	+278,3%
Duplicação de código	34%	3%	91,2%
Violações SOLID	47	2	95,7%

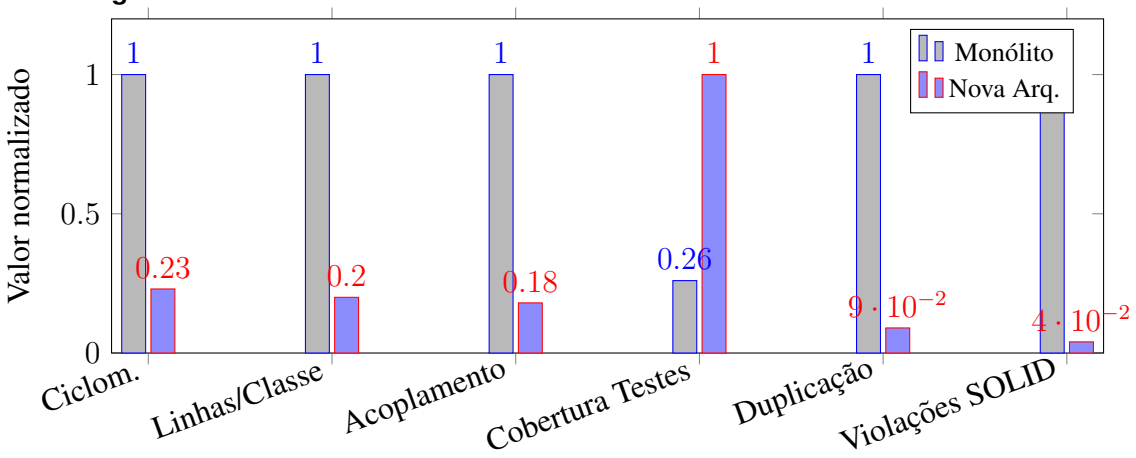
A organização em camadas e o uso de contratos (*interfaces*) reduziramsignificamente o *time-to-productivity* de desenvolvedores iniciantes, em consonância com a literatura sobre a curva de aprendizado em projetos complexos [**?**]. A Tabela 5 resume ganhos observados.

Tabela 5. Produtividade e onboarding (resumo observacional)

Métrica	MVC Trad.	Clean Arch.	Ganho
1º commit (dias)	3–5	1–2	50–67%
Use cases/semana	1–2	3–4	100–200%
Bugs por funcionalidade	3–5	1–2	60–75%
Tempo p/ localizar código (min)	15–30	2–5	80–90%
Mentoria (h/semana)	8–10	2–3	70–80%

Embora a migração introduza maior complexidade operacional com necessidade

Figura 4. Comparativo visual (normalizado) de indicadores de qualidade de código.



de orquestração de serviços e mensageria, os ganhos substanciais em performance e escalabilidade superam amplamente o custo adicional. A combinação de melhoria de throughput (7.042.800%) e redução de latência (99,74%) resulta em *ROI* técnico highly positivo na operação e evolução da plataforma.

As principais limitações incluem a complexidade operacional, que demanda práticas maduras de DevOps e observabilidade distribuída, e a consistência eventual típica de fluxos assíncronos, exigindo definição rigorosa de SLOs e mecanismos de tolerância (idempotência, retries e filas de dead-letter). Há ainda o sobrecusto de rede introduzido pela mensageria, compensado pelo maior desacoplamento entre serviços. Como trabalhos futuros, propõe-se automatizar a análise estática hoje manual (p.ex., com SonarQube/Code Climate integrados ao CI), além de ampliar testes (incluindo caos e tracing de ponta a ponta) e calibrar filas e timeouts para cenários de maior escala.

Referências

Arifin, J. and Perdana, R. S. (2019). Ugrade: Autograder for competitive programming using contestant pc as worker. In *2019 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6. IEEE.

Beck, K., Fowler, M., and Beck, G. (1999). Bad smells in code. *Refactoring: Improving the design of existing code*, 1(1999):75–88.

Cortes, P. V. C. et al. (2019). Programação apoiada por ambientes virtuais e juízes online no ensino semipresencial. *Caderno de Graduação-Ciências Exatas e Tecnológicas-UNIT-SERGIPE*, 5(2):113–113.

de Campos, C. P. and Ferreira, C. E. (2004). Boca: um sistema de apoio a competições de programação. In *Workshop de Educação em Computação*. Sociedade Brasileira de Computação.

do Nascimento Ponciano, K. M., Silva Filho, A. A., Bourguet, J.-R., and de Oliveira, E. (2024). Creating an academic prometheus in brazil: Weaving check50, autolab and moss into a unified autograder. In *CSEDU (2)*, pages 439–450.

Estdale, J. and Georgiadou, E. (2018). Applying the iso/iec 25010 quality models to software product. In *European Conference on Software Process Improvement*, pages 492–503. Springer.

Esteban Velasco, L., Trillo Carreras, J., and Burgos Sosa, R. (2024). Code analysis and instrumentation of submissions for online judges.

Gunther, N. J. (2007). *Guerrilla capacity planning: a tactical approach to planning for highly scalable applications and services*. Springer.

Liu, K., Han, Y., Zhang, J. M., Chen, Z., Sarro, F., Harman, M., Huang, G., and Ma, Y. (2023). Who judges the judge: An empirical study on online judge tests. In *Proceedings of the 32nd acm sigsoft international symposium on software testing and analysis*, pages 334–346.

Lopes, J. F. A., Lima, M. V. A., da Cruz Santos, C., and Orbolato, D. R. S. (2025). Extensao universitária em açao: Experiências na disseminaçao do conhecimento em programaçao. In *Workshop sobre Educação em Computação (WEI)*, pages 825–836. SBC.

Martin, R. C. (2017). *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press.

Mekterović, I., Brkić, L., and Horvat, M. (2023). Scaling automated programming assessment systems. *Electronics*, 12(4):942.

Nielsen, J. (1994). *Usability engineering*. Morgan Kaufmann.

Pea, R. D. and Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New ideas in psychology*, 2(2):137–168.

Peveler, M., Maicus, E., and Cutler, B. (2019). Comparing jailed sandboxes vs containers within an autograding system. In *Proceedings of the 50th ACM technical symposium on computer science education*, pages 139–145.

Pham, M. T. and Nguyen, T. B. (2019). The domjudge based online judge system with plagiarism detection. In *2019 IEEE-RIVF International Conference on Computing and Communication Technologies (RIVF)*, pages 1–6. IEEE.

Preuss, J. O. and de Lima, C. C. (2023). Ferramentas online na aprendizagem de programação de computadores no contexto do ensino remoto. *Revista Brasileira de Informática na Educação*, 31:790–813.

Toledo, R. Y. and Mota, Y. C. (2014). An e-learning collaborative filtering approach to suggest problems to solve in programming online judges. *International Journal of Distance Education Technologies (IJDET)*, 12(2):51–65.

Tostes, R. N. and Sirqueira, T. F. M. (2023). O processo de migração de um monolito para uma arquitetura de microsserviços utilizando serverless. *Caderno de Estudos em Engenharia de Software*, 4(2).

Williams, J. (2012). *RabbitMQ in action: distributed messaging for everyone*. Simon and Schuster.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., Wesslén, A., et al. (2012). *Experimentation in software engineering*, volume 236. Springer.