EEVEE: Uma Arquitetura Extensível Baseada em Workers para Avaliação Automatizada de Programação

João Vitor Coimbra
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET/RJ)
Rio de Janeiro, Brasil
joao.coimbra@aluno.cefet-rj.br

Diego Castro
Centro Federal de Educação Tecnológica Celso Suckow da Fonseca (CEFET/RJ)
Rio de Janeiro, Brasil
diego.castro@cefet-rj.br

Resumo

Ferramentas de avaliação automatizada de exercícios de programação costumam estar acopladas a uma linguagem ou a uma cadeia de ferramentas específica, o que dificulta sua evolução para novos contextos de ensino. Este artigo de ferramenta descreve, do ponto de vista arquitetural, o mecanismo de execução do EEVEE, uma plataforma de avaliação automatizada que desacopla a execução do código submetido da linguagem de programação por meio de imagens de contêiner. Cada submissão é executada em um *job* efêmero e isolado, orquestrado por Kubernetes, no qual um contêiner de inicialização padronizado injeta o código do estudante em um volume compartilhado e um contêiner principal, específico da tecnologia, executa os testes e escreve o resultado em sua saída padrão, que a plataforma converte em um resultado estruturado comum. Como o núcleo de orquestração manipula apenas contêineres opacos, o suporte a uma nova tecnologia reduz-se a construir uma imagem que cumpra um contrato mínimo e a registrar a estratégia que interpreta sua saída, sem alterar o *bootstrap*, o ciclo de vida do *job*, a coleta de *logs* nem o esquema comum de resultados. Descrevemos esse contrato, o catálogo atual de imagens e o caminho de extensão, e disponibilizamos a ferramenta em repositório público e em uma instância acessível.

Palavras-chave

Avaliação Automatizada, Contêineres, Kubernetes, Arquitetura de Software, Extensibilidade, Isolamento de Execução

1 Introdução

A avaliação automatizada de exercícios é um recurso consolidado no ensino de programação, com forte impacto na redução da carga de correção e na oferta de retorno rápido aos estudantes [3, 12, 13]. Muitas dessas ferramentas, contudo, nascem atreladas a uma única linguagem ou a uma cadeia de ferramentas específica: a lógica de execução dos testes é entrelaçada com o interpretador, o compilador e o *runner* de uma tecnologia particular. Esse acoplamento limita a evolução da plataforma, pois a inclusão de uma nova linguagem tende a exigir modificações no seu núcleo.

Este artigo de ferramenta descreve o mecanismo de execução do EEVEE, uma plataforma de avaliação automatizada de programação [8], com foco em uma decisão arquitetural específica: *desacoplar a execução do código submetido da linguagem de programação por meio de imagens de contêiner*. A execução de cada submissão ocorre em um *job* efêmero e isolado, orquestrado por Kubernetes [1, 2], em que a única fronteira entre a plataforma e a tecnologia avaliada é uma imagem de contêiner que cumpre um contrato mínimo. Em consequência, o suporte a uma nova tecnologia não requer alteração do núcleo de orquestração: reduz-se a construir uma imagem que respeite o contrato e a registrar a estratégia que interpreta sua saída. Do ponto de vista de evolução e manutenção de software, essa fronteira localiza as mudanças tecnológicas: requisitos de *runner*, dependências, *parsing* e permissões ficam confinados ao *worker* e à sua estratégia, enquanto o núcleo de injeção, orquestração e coleta permanece estável.

As contribuições deste trabalho são: (i) a descrição arquitetural do modelo de execução baseado em imagens de contêiner e do contrato de tempo de execução que localiza a variação tecnológica e o torna agnóstico à linguagem; (ii) a caracterização das camadas de isolamento e de segurança que cercam a execução de código não confiável, incluindo a restrição de rede por perfil componível; (iii) o catálogo de imagens de *worker* atualmente disponíveis, ilustrado por um exemplo de interação segura com um provedor de nuvem; e (iv) uma análise estrutural da localização de mudanças em três *workers* heterogêneos, complementada pela discussão do caminho de extensão a novas linguagens. A ferramenta é disponibilizada como software de código aberto, em repositório persistente e em instância pública.

O restante do artigo posiciona o trabalho frente à literatura (Seção 2), detalha a arquitetura de execução e suas camadas de segurança (Seção 3), apresenta o catálogo de *workers*, a evidência de localização de mudanças e o caminho de extensão (Seções 4–6) e, por fim, discute limitações (Seção 7) e conclui (Seção 8).

2 Contexto e Trabalhos Relacionados

A avaliação automatizada de programação é apoiada por um amplo conjunto de ferramentas e abordagens, revisadas em diversos estudos secundários [3, 12, 13]. Sistemas de *online judge*, amplamente empregados em competições e em disciplinas introdutórias, avaliam submissões majoritariamente por comparação entre entrada e saída esperada e, em geral, oferecem suporte a um conjunto de linguagens fixo, embutido no próprio juiz [19]. Esse acoplamento entre o mecanismo de avaliação e as linguagens suportadas dificulta a incorporação de novas tecnologias, sobretudo quando a correção depende de *frameworks* de teste ricos, e não apenas da verificação de saída.

No nível do exercício, *frameworks* como o TESTed propõem suítes de teste agnósticas à linguagem, descrevendo os casos de teste de maneira independente da tecnologia empregada pelo estudante [17]. O EEVEE situa-se em um nível complementar a essas propostas: em vez de padronizar a descrição dos testes, padroniza o *ambiente de execução*. Cada tecnologia é encapsulada em uma imagem de contêiner que cumpre um contrato mínimo, de modo que o mesmo

Tabela 1: Comparação com ferramentas de avaliação baseadas em contêineres ou *sandboxes*.

Ferramenta	Extensão tecnológica	Isolamento	Perfil de segurança
INGInious	Contêiner/agente por tarefa	Docker	Limites de recursos
CodeRunner	Tipo de questão no *sandbox* Jobe	Jobe	Limites do Jobe
PrairieLearn	Imagem de *grader* por questão	Docker	Sem rede (padrão)
Judge0	Linguagem pré-configurada	`isolate`	Tempo e memória
EEVEE	Imagem + estratégia registrada	*Job* K8s efêmero	Perfis componíveis

mecanismo de orquestração, injeção de código e coleta de resultados serve a qualquer linguagem, inclusive àquelas que exigem um *runner* de testes específico ou um passo de compilação. Essa escolha desloca a variação de linguagem para dentro da imagem, mantendo o núcleo da plataforma estável.

Mais próximas do EEVEE estão as ferramentas que executam submissões em contêineres ou *sandboxes* isolados. O INGInious avalia exercícios por meio de agentes que executam contêineres Docker, com limites de recursos configuráveis por tarefa [5]. O CodeRunner, integrado ao Moodle, delega a execução ao *sandbox* Jobe e organiza a variação por tipo de questão [11]. O PrairieLearn admite *external graders*, corretores fornecidos como imagens Docker customizadas e executados, por padrão, sem acesso à rede [20]. O Judge0 oferece execução modular e isolada para múltiplas linguagens pré-configuradas, apoiando-se no isolador `isolate` e em limites de tempo e memória [6]. A Tabela 1 contrasta essas abordagens segundo quatro eixos. O diferencial do EEVEE não está em usar contêineres como unidade de extensão — prática compartilhada com INGInious e PrairieLearn —, mas na *combinação* de imagem mais estratégia registrada, normalização de resultado externa ao *runner* e perfis de segurança *componíveis* por atividade, sobre orquestração efêmera em Kubernetes.

3 Arquitetura de Execução

A plataforma separa a interação com usuários e a persistência de dados, de um lado, da execução de código não confiável, de outro. Este artigo concentra-se na segunda parte. Quando uma submissão precisa ser avaliada, o serviço de agendamento cria dinamicamente um *job* do Kubernetes (`batch/v1`) dedicado àquela execução. A Figura 1 resume o fluxo.

Contrato de tempo de execução. O elemento que torna a execução agnóstica à linguagem é um contrato mínimo, cumprido por qualquer imagem de *worker*, com três pontos convencionados. O *ponto de injeção* determina onde o código chega: o código submetido pelo estudante e os testes elaborados pelo docente são materializados em um volume efêmero (`emptyDir`) montado em diretórios convencionados (`/app/src` e `/app/test`). O *ponto de entrada* determina como a avaliação começa: a imagem expõe um comando padrão que, se necessário, compila o código e, em seguida, executa a suíte de testes. O *ponto de saída* determina como o resultado é comunicado: ao final, o contêiner escreve em *stdout* o desfecho da execução dos testes. O contrato não impõe um formato de texto

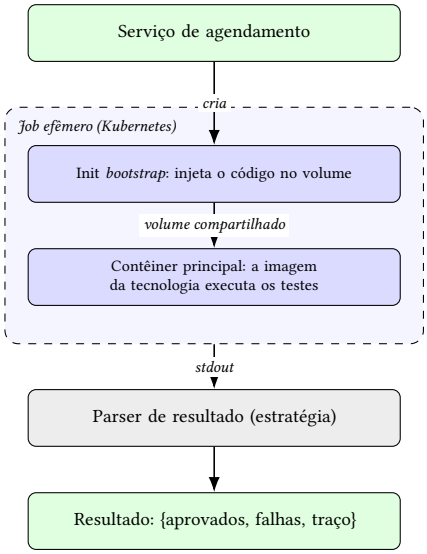

Figura 1: Fluxo de execução de uma submissão. O contêiner de inicialização injeta o código em um volume compartilhado; o contêiner principal, específico da tecnologia, executa os testes e emite um resumo em *stdout*, que a estratégia correspondente converte em um resultado estruturado.

Código 1: Descrição de tarefa (`WorkerDefinition`) entregue ao contêiner de inicialização.

```
type WorkerDefinition = {
  srcPath: string;   // ex.: "/app/src"
  testPath: string;  // ex.: "/app/test"
  files: Record<string, string> | null;      // codigo do estudante
  testFiles: Record<string, string> | null;  // testes do docente
  dependencies: string[];                     // pacotes a instalar
};
```

Código 2: Exemplo de saída em *stdout* de um *worker* baseado em Jest; a estratégia correspondente a interpreta.

```
Tests:       4 passed, 4 total
Failed:      0
ResultsFile: /app/test-results.json
Tests run!
```

único a esse ponto: cada *runner* pode emitir a sua própria saída, cabendo à plataforma convertê-la em um resultado estruturado comum, conforme detalhado adiante. O Código 1 apresenta a estrutura injetada e o Código 2, um exemplo de saída de um *worker* baseado em Jest.

Nenhum desses três pontos menciona uma linguagem específica: eles fixam apenas caminhos, um comando e um fluxo de texto em *stdout*, e é essa neutralidade que permite reaproveitar todo o restante da plataforma para tecnologias distintas.

Injeção do código. A submissão não é embutida na imagem, o que permite que a mesma imagem sirva a inúmeras submissões. Um contêiner de inicialização compartilhado, o *worker-bootstrap*, recebe a `WorkerDefinition` serializada e codificada em *base64* por

uma variável de ambiente, o que evita os limites de tamanho impostos a argumentos e dispensa volumes de configuração externos. Ele decodifica a descrição, normaliza os caminhos para impedir escritas fora dos diretórios previstos (evitando travessia de diretório) e materializa os arquivos no volume compartilhado antes que o contêiner principal inicie. Como esse contêiner apenas escreve arquivos, ele é único para todas as tecnologias; a lógica específica de cada linguagem reside exclusivamente na imagem principal.

Orquestração do job. O fluxo da Figura 1 é concretizado por um manifesto de *job* do Kubernetes, cujo esqueleto aparece no Código 3. Nele, o contêiner de inicialização e o contêiner principal compartilham um volume `emptyDir` montado nos diretórios do contrato; os campos `backoffLimit`, `restartPolicy` e `automountServiceAccountToken` materializam as garantias de execução única e de privilégio mínimo discutidas na Seção 3.1.

Código 3: Esqueleto do manifesto do *job* (campos abreviados).

```
apiVersion: batch/v1
kind: Job
spec:
  backoffLimit: 0
  template:
    spec:
      restartPolicy: Never
      automountServiceAccountToken: false
      initContainers:
        - name: worker-bootstrap       # injeta o codigo
          env: [{name: WORKER_DEFINITION_B64, value: <base64>}]
          volumeMounts:
            - {name: app, mountPath: /app/src,  subPath: src}
            - {name: app, mountPath: /app/test, subPath: test}
      containers:
        - name: worker                 # imagem da tecnologia
          command: ["/bin/sh", "-c", "npm start"]
          resources: {limits: {cpu: "1", memory: 512Mi}}
      volumes:
        - {name: app, emptyDir: {}}
```

Execução e coleta do resultado. Iniciado o contêiner principal, seu ponto de entrada dispara o *runner* de testes sobre os arquivos injetados e escreve o desfecho em *stdout*. A plataforma acompanha o ciclo de vida do *job* e, após o encerramento, recupera os registros (*logs*) do *pod*. É aqui que ocorre a padronização: cada tipo de *worker* possui uma *estratégia* que conhece o formato de saída do seu *runner* e o converte em um resultado estruturado comum, contendo o número de aprovados, o número de falhas e o traço completo. A estratégia é, portanto, o único componente que lida com a diversidade de formatos; o orquestrador manipula apenas o resultado estruturado e permanece indiferente à tecnologia avaliada. *Workers* construídos sobre o mesmo *runner* compartilham a mesma estratégia; por exemplo, as imagens baseadas em Jest reutilizam um único analisador, enquanto as baseadas em Cypress usam outro.

3.1 Segurança em Camadas

Como o código submetido é potencialmente não confiável, o modelo de execução aplica múltiplas camadas de defesa independentes, ilustradas na Figura 2, seguindo os princípios de defesa em profundidade e de privilégio mínimo [15]. Na base, cada submissão executa em um contêiner isolado [14, 16], dentro de um *job* efêmero descartado após a execução (`restartPolicy` `Never` e `backoffLimit` zero), sem credenciais do plano de controle (`automountServiceAccountToken` desabilitado) e sob limites explícitos de CPU e memória. Na camada de rede, os *workers* têm, por padrão, o tráfego de saída

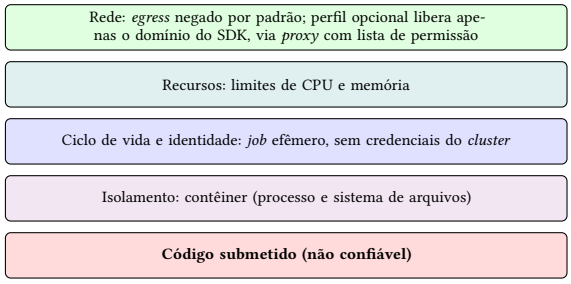

Figura 2: Camadas de defesa em profundidade que envolvem a execução do código não confiável. Cada camada é independente; o perfil de rede é ativado apenas para *workers* que exigem acesso externo legítimo.

negado (*default-deny egress*) e não alcançam os serviços internos da plataforma.

Alguns exercícios, porém, exigem acesso externo legítimo. A trilha construída para o estudo comparativo do ORM analítico TeraORM, por exemplo, precisa consultar o Google BigQuery. Em vez de simplesmente abrir a rede, a plataforma associa a esse *worker* um *perfil de rede* (um rótulo `eevee/network-profile`) que ativa uma política de rede (*NetworkPolicy* [10]) e encaminha todo o tráfego de saída por um *proxy* local. Esse *proxy* mantém uma lista de permissão restrita aos domínios de API do Google, negando qualquer outro destino. Assim, o código do estudante alcança exclusivamente o serviço necessário à atividade, e nada além dele. Como o perfil é apenas mais um rótulo aplicado pela estratégia do *worker*, a postura de segurança torna-se composável: cada tecnologia adiciona precisamente as permissões que requer. O Código 4 resume esse controle em dois níveis: uma *NetworkPolicy* restringe o destino ao *proxy* e o *proxy* libera apenas o domínio do SDK.

Código 4: Controle de *egress* em dois níveis para o perfil `google-egress`.

```
# 1) NetworkPolicy: saida apenas para o proxy (e DNS)
kind: NetworkPolicy
spec:
  podSelector: {matchLabels: {eevee/network-profile: google-egress}}
  policyTypes: [Egress]
  egress:
    - to: [{podSelector: {matchLabels: {app: eevee-egress-proxy}}}]
# 2) Proxy (Squid): apenas dominios de API do Google
acl allowed_google dstdomain .googleapis.com
http_access allow allowed_google
http_access deny all
```

4 Catálogo de Workers e Extensibilidade

O contrato descrito é atendido, hoje, por um conjunto de imagens que cobre diferentes tecnologias e estilos de teste, apresentado na Tabela 2. A variedade, que inclui testes unitários com Jest [9], testes de ponta a ponta com Cypress [4], serviços gRPC e cenários com banco de dados relacional, demonstra que o modelo de execução acomoda contextos heterogêneos sob a mesma orquestração.

Segundo o contrato arquitetural, incluir uma linguagem ainda não contemplada exigiria duas ações localizadas, sem mudanças previstas na lógica genérica de orquestração. Primeiro, constrói-se uma imagem que cumpra o contrato: um `Dockerfile` a partir da imagem base da linguagem, um ponto de entrada que execute o

Tabela 2: Imagens de *worker* disponíveis e suas tecnologias.

Imagem de *worker*	Tecnologia / execução de testes
`node-default`	Node.js, testes unitários (Jest)
`node-nestjs`	NestJS, testes de integração (Jest)
`react-cypress`	React, testes de ponta a ponta (Cypress)
`nextjs-cypress`	Next.js, testes de ponta a ponta (Cypress)
`grpc`	Serviços gRPC em Node.js
`node-teraorm`	Node.js com acesso a *data warehouse*
`*-postgresql`	Variantes com banco PostgreSQL anexo

runner de testes sobre `/app/src` e `/app/test`, e a emissão do resumo padronizado em *stdout*. Segundo, registra-se um novo tipo de *worker* e a *estratégia* que interpreta a saída daquele *runner*, o que atualiza o registro declarativo do agendador. Permanecem inalterados o contêiner de inicialização, o esquema de injeção por volume, o *manifesto* do *job* e o mecanismo de coleta, que constituem o núcleo estável da arquitetura.

Esse registro, mantido em um único ponto do agendador, associa cada tipo de *worker* a uma imagem e a uma *estratégia* que implementa uma interface comum: declara o nome da imagem, os diretórios do contrato e as funções que constroem o comando, montam a `WorkerDefinition` e interpretam a saída do *runner*. As duas seções seguintes examinam casos complementares: a Seção 5 detalha uma imagem do catálogo que interage com um provedor de nuvem, e a Seção 6 percorre a adição de uma linguagem compilada, o Go.

4.1 Análise Estrutural da Localização de Mudanças

Para ir além da descrição arquitetural, examinamos a estrutura de três *workers* do catálogo com requisitos deliberadamente distintos e verificamos *onde* cada variação se aloja no código. Os *workers* foram selecionados para maximizar a variação em três dimensões: o *runner* de testes, o estilo de execução (unitário *versus* ponta a ponta) e a necessidade de acesso externo. A análise incidiu sobre o estado do repositório etiquetado como `v1.0-vem2026`, considerando como núcleo genérico os módulos de inicialização (*worker-bootstrap*), de construção do *job* e de coleta de resultados, e contabilizando as linhas não vazias e não comentadas de cada *trigger* de imagem e classe de estratégia. A Tabela 3 resume o achado: as adições concentram-se na imagem e na estratégia, e os três *workers* produzem o mesmo resultado comum (`{passes, failures, trace}`).

A inclusão de um *worker modifica* o registro declarativo do agendador — o *enum* `WorkerType`, o mapa `strategyByWorkerType` e o prefixo de *job* em `worker.constants.ts` —, mas não altera os componentes genéricos responsáveis pelo *bootstrap*, pela construção do *job*, pelo acompanhamento do ciclo de vida, pela coleta de *logs* ou pelo esquema de resultados. O caso do `node-teraorm` é ilustrativo: seu *trigger* de imagem é idêntico ao do `node-default`, e todo o acréscimo tecnológico (credencial, perfil de rede e *proxy*) reside no método que compõe as opções do *job* de sua estratégia, sem tocar o fluxo comum.

Cenário reproduzível. Para verificação de ponta a ponta sem um *cluster*, o alvo `make reproduce-node-default` constrói a imagem `node-default`, injeta o exemplo determinístico de `examples/reproduce/` e executa a suíte em um contêiner efêmero, emitindo

Tabela 3: Localização das mudanças em três *workers* heterogêneos. As adições específicas confinam-se à imagem e à estratégia; a inclusão atualiza o registro declarativo do agendador, mas não a lógica genérica de orquestração. As contagens referem-se às linhas não vazias e não comentadas do *trigger* da imagem e da classe de estratégia.

Worker	Adições específicas da tecnologia	Lógica genérica alterada?
`node-default`	Imagem Node e *trigger* Jest (61 l.); estratégia com `parseJestLogResult` (84 l.)	Não
`react-cypress`	Imagem `cypress/browsers` e *trigger* Cypress (164 l.); estratégia com *parser* de envelope (35 l.)	Não
`node-teraorm`	Reusa a imagem e o *trigger* Jest do `node-default`; a estratégia (163 l.) adiciona dependências pré-carregadas, *secret*, o rótulo `network-profile` e o *proxy* de *egress*	Não

a saída do Código 2, que a estratégia converte em `{passes: 4, failures: 0}`. Cabe delimitar o alcance desta análise: ela demonstra a *localização de mudanças* entre tecnologias e *runners* distintos, mas ainda não valida empiricamente a inclusão de uma linguagem fora do ecossistema JavaScript/TypeScript, discutida de forma prospectiva na Seção 6.

5 Interação com Provedores de Nuvem: a Imagem para BigQuery

Entre as imagens do catálogo (Tabela 2), a designada `node-teraorm` ilustra de forma concreta o alcance do modelo de execução. Enquanto muitos exercícios se restringem a algoritmos autocontidos, há valor pedagógico em atividades que interagem com serviços reais, como um *data warehouse* na nuvem. Essa imagem permite ao estudante consultar o Google BigQuery por meio do SDK nativo do provedor e é ilustrativa porque reúne, sob o mesmo contrato de execução da Seção 3, três necessidades que uma atividade "de nuvem" impõe, todas confinadas à estratégia do *worker*, sem tocar o núcleo. Primeiro, **dependências**: a imagem parte da mesma base Node e é pré-carregada com o SDK do BigQuery e bibliotecas de consulta, mantendo o ponto de entrada inalterado. Segundo, **credenciais efêmeras**: em vez de embuti-la na imagem, a plataforma entrega a credencial de conta de serviço como um *secret* do Kubernetes, montado apenas no *job* daquela execução (via `GOOGLE_APPLICATION_CREDENTIALS`) e descartado com ele. Terceiro, **acesso externo controlado**: a imagem ativa o perfil de rede `google-egress` da Seção 3.1 (Código 4), que bloqueia toda a saída exceto um *proxy* restrito ao domínio de API do Google, para onde a variável `HTTPS_PROXY` encaminha o tráfego do SDK.

O restante (injeção do código, coleta do resultado e as demais camadas de isolamento) é idêntico ao de qualquer outro *worker*. Assim, o mesmo modelo de execução que avalia algoritmos simples sustenta atividades de engenharia de dados e de consumo de APIs de nuvem, sem abrir mão do isolamento nem exigir mudanças no núcleo da plataforma.

6 Evolução: Extensão a Linguagens Compiladas

O catálogo atual concentra-se no ecossistema JavaScript/TypeScript, mas o contrato de execução não pressupõe uma linguagem interpretada. Para tornar concreto o caminho de extensão a uma linguagem fora desse ecossistema, esta seção descreve, de forma *ilustrativa*, como o suporte a Go, uma linguagem compilada com cadeia de ferramentas e formato de saída próprios, poderia ser incorporado. Trata-se de um exercício de projeto, não do relato de uma imagem publicada: o objetivo é evidenciar que a arquitetura acomodaria uma linguagem compilada exercitando apenas os pontos de extensão previstos, sem tocar o núcleo.

A **imagem** partiria de uma base `golang` e seu ponto de entrada executaria `go test` sobre o código injetado em `/app/src` e `/app/test`. O passo de compilação, ausente nas imagens Node, é absorvido pelo próprio `go test`, de modo que o contrato permaneceria inalterado, pois exige apenas que o ponto de entrada compile (se necessário), execute os testes e escreva o desfecho em *stdout*. O Código 5 esboça esse ponto de entrada, que normalizaria a saída de `go test -json` [18] no mesmo resumo textual dos *workers* Node. Do lado do orquestrador, bastaria **registrar** um novo tipo de *worker* e uma *estratégia* para interpretar essa saída, atualizando o registro declarativo; segundo o contrato arquitetural, todo o restante (contêiner de inicialização, injeção por volume, manifesto do *job*, coleta de registros e camadas de segurança) seria reutilizado sem mudanças previstas na lógica genérica de orquestração. O mesmo roteiro aplicar-se-ia a Python, com `pytest`, e a linguagens da JVM.

Código 5: Esboço do ponto de entrada de um *worker* Go: compila e executa os testes e escreve o desfecho em *stdout*.

```
// executa os testes sobre o codigo injetado
out, _ := exec.Command("go", "test", "./...", "-json").Output()
passed, total := parseGoTestJSON(out) // conta acoes pass/fail
fmt.Printf("Tests:       %d passed, %d total\n", passed, total)
fmt.Printf("Failed:      %d\n", total-passed)
fmt.Println("Tests run!")
```

Direções futuras. Planeja-se distribuir as imagens por um *registry* de contêineres, permitindo que instituições publiquem e compartilhem *workers* sem reconstruí-los localmente.

7 Limitações

O modelo pressupõe que cada tecnologia disponha de um *runner* de testes capaz de emitir um resumo textual; linguagens sem tal ferramenta exigiriam um adaptador adicional embutido na imagem. A criação de um *job* por submissão introduz latência de inicialização (*cold start*), mitigável pelo reaproveitamento de imagens em cache no nó, mas relevante sob picos de submissão simultânea. Por fim, a orquestração assume a disponibilidade de um *cluster* Kubernetes; implantações de menor porte podem recorrer a distribuições locais, ao custo de menor elasticidade.

8 Considerações Finais

Este artigo descreveu, do ponto de vista arquitetural, como o EEVEE torna a execução de código agnóstica à linguagem por meio de imagens de contêiner e de um contrato mínimo de tempo de execução, confinando a linguagem, o formato de resultado e a postura de segurança a pontos de extensão bem delimitados. O catálogo atual concentra-se no ecossistema JavaScript/TypeScript, mas, como discutido na Seção 6, o contrato arquitetural indica que estender a plataforma a linguagens compiladas, como Go, ou interpretadas, como Python, exigiria uma nova imagem, uma nova estratégia e a atualização do registro do agendador, sem mudanças previstas na lógica genérica de orquestração. Essa separação clara entre o que é estável e o que é extensível favorece a evolução sustentável da ferramenta e sua adoção em contextos de ensino heterogêneos.

Disponibilidade de Artefatos

O código-fonte aberto do EEVEE está disponível em [8], e sua instância pública em [7].

Referências

[1] Brendan Burns, Joe Beda, and Kelsey Hightower. 2019. *Kubernetes: Up and Running* (2 ed.). O'Reilly Media.

[2] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57. doi:10.1145/2890784

[3] Sébastien Combéfis. 2022. Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools. *Software* 1, 1 (2022), 3–30. doi:10.3390/software1010002

[4] Cypress. 2025. Cypress: JavaScript End-to-End Testing Framework. https://www.cypress.io/. Acesso em: 08 jul. 2026.

[5] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. 2015. Automatic grading of programming exercises in a MOOC using the INGInious platform. In *Proceedings of the European MOOCs Stakeholders Summit (EMOOCs)*. 86–91.

[6] Herman Zvonimir Došilović and Igor Mekterović. 2020. Robust and Scalable Online Code Execution System. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, 1627–1632. doi:10.23919/MIPRO48935.2020.9245310

[7] EEVEE Project. 2026. EEVEE: Instância Pública. https://eevee.testbeds.rnp.br/. Instância implantada em infraestrutura da RNP. Acesso em: 08 jul. 2026.

[8] EEVEE Project. 2026. EEVEE: Plataforma de Avaliação Automatizada de Programação. https://github.com/COCSI-MG/eevee. Repositório do código-fonte. Acesso em: 08 jul. 2026.

[9] Jest. 2025. Jest: Delightful JavaScript Testing. https://jestjs.io/. Acesso em: 08 jul. 2026.

[10] Kubernetes Documentation. 2025. Network Policies. https://kubernetes.io/docs/concepts/services-networking/network-policies/. Acesso em: 10 jul. 2026.

[11] Richard Lobb and Jenny Harlow. 2016. Coderunner: A Tool for Assessing Computer Programming Skills. *ACM Inroads* 7, 1 (2016), 47–51. doi:10.1145/2810041

[12] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaojing Shi. 2024. Automated Grading and Feedback Tools for Programming Education: A Systematic Review. *ACM Transactions on Computing Education* 24, 1 (2024). doi:10.1145/3636515

[13] José Paiva, José Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education* 22, 3 (2022). doi:10.1145/3513140

[14] Allison Randal. 2020. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. *Comput. Surveys* 53, 1 (2020). doi:10.1145/3365199

[15] Jerome H. Saltzer and Michael D. Schroeder. 1975. The Protection of Information in Computer Systems. *Proc. IEEE* 63, 9 (1975), 1278–1308. doi:10.1109/PROC.1975.9939

[16] Sachchidanand Singh and Nirmala Singh. 2016. Containers & Docker: Emerging Roles & Future of Cloud Technology. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*. IEEE, 804–807. doi:10.1109/ICATCCT.2016.7912109

[17] Niko Strijbol, Charlotte Van Petegem, Rien Maertens, Boris Sels, Christophe Scholliers, Peter Dawyndt, and Bart Mesuere. 2023. TESTed — An educational testing framework with language-agnostic test suites for programming exercises. *SoftwareX* 22 (2023), 101404. doi:10.1016/j.softx.2023.101404

[18] The Go Authors. 2025. Package testing — The Go Programming Language. https://pkg.go.dev/testing. Acesso em: 10 jul. 2026.

[19] Yutaka Watanobe, Md. Mostafizer Rahman, Taku Matsumoto, Uday Kiran Rage, and Penugonda Ravikumar. 2022. Online Judge System: Requirements, Architecture, and Experiences. *International Journal of Software Engineering and Knowledge Engineering* 32, 6 (2022), 917–946. doi:10.1142/S0218194022500346

[20] Matthew West, Geoffrey L. Herman, and Craig Zilles. 2015. PrairieLearn: Mastery-based Online Problem Solving with Adaptive Scoring and Recommendations Driven by Machine Learning. In *2015 ASEE Annual Conference & Exposition*. ASEE, 26.1238.1–26.1238.14. doi:10.18260/p.24575