Uma Plataforma Multimodelo para Governança e Compartilhamento de Inferência em Ferramentas Inteligentes de Engenharia de Software

Jeosafá de Melo Freitas Júnior
VIRTUS/UFCG
Campina Grande, Brasil
jota.freitas@virtus.ufcg.edu.br

Joaquim José Cintra Maia Honório
VIRTUS/UFCG
Campina Grande, Brasil
joaquim.honorio@virtus.ufcg.edu.br

Danyllo Albuquerque
VIRTUS/UFCG
Campina Grande, Brasil
danyllo.albuquerque@virtus.ufcg.edu.br

Emanuel Dantas Filho
UFCG
Campina Grande, Brasil
emanueldan@gmail.com

Resumo

Ferramentas inteligentes de apoio ao desenvolvimento de software, entre elas assistentes de programação, análise de código, busca com geração aumentada por recuperação (RAG) e agentes de investigação, dependem de modelos de linguagem para operar. Quando processam código proprietário ou dados sob sigilo, o envio de *prompts* a provedores externos torna-se inviável, uma vez que cada fornecedor retém as entradas por prazos próprios e subtrai à organização o controle sobre o trânsito de código e dados. A inferência passa, então, a ocorrer na infraestrutura da própria organização, o que restringe as opções aos modelos de pesos abertos e converte a GPU em um recurso finito a ser planejado. Na ausência de uma camada comum, cada sistema tende a provisionar uma cópia própria de cada modelo, o que duplica o consumo de capacidade e fragmenta o controle de acesso. Este artigo descreve uma plataforma de inferência multimodelo que expõe os modelos sob uma única API, reparte uma capacidade de GPU compartilhada e estrutura o acesso em uma hierarquia de núcleos, projetos, usuários e chaves, preservando a atribuição de cada requisição à sua origem e permitindo que uma única cópia de cada modelo atenda a todos os sistemas. Como a centralização introduz uma camada adicional no caminho de cada requisição, avalia-se o sobrecusto de latência dela decorrente frente ao acesso direto ao servidor de inferência, por meio de um experimento pareado. Situado no domínio de MLOps, o trabalho sistematiza os princípios de engenharia subjacentes, examina as dificuldades encontradas e aponta direções para equipes diante de desafios semelhantes.

Palavras-chave

Modelos de linguagem, Governança de IA, Plataforma de inferência

1 Introdução

Modelos generativos passaram a apoiar várias atividades de engenharia de software. Além da geração de código, esses modelos são utilizados na criação de testes, análise de falhas, revisão de segurança, recuperação de documentação e manutenção automatizada de repositórios [1–3]. Ferramentas dessa classe frequentemente enviam aos modelos trechos de código, mensagens de erro, requisitos, resultados de testes e documentos internos.

A forma mais simples de integrar essas ferramentas é consumir *APIs* de provedores externos. Essa alternativa transfere ao provedor a operação dos modelos, mas também faz com que código, *prompts* e documentos trafeguem por infraestrutura externa. As políticas de retenção e utilização dos dados variam conforme o provedor e o contrato adotado [4, 5].

Esse modelo pode ser inadequado quando a ferramenta processa código proprietário, propriedade intelectual ou informações sujeitas a regras internas de acesso. Nesses casos, uma organização pode optar por executar modelos em sua própria infraestrutura. A inferência local amplia o controle sobre o fluxo de dados, mas cria problemas de engenharia: os modelos precisam caber no hardware disponível, as *GPUs* passam a ser recursos finitos, diferentes aplicações disputam a mesma capacidade e cada chamada precisa ser autenticada, autorizada e atribuída ao projeto responsável.

Sem uma camada comum, cada ferramenta tende a manter seu próprio servidor, sua própria cópia dos modelos e mecanismos independentes de credenciais e observabilidade. Essa fragmentação repete trabalho operacional, reserva memória de GPU de forma redundante e distribui políticas de acesso entre vários sistemas.

Ferramentas inteligentes de engenharia de software apresentam requisitos operacionais que diferem daqueles de aplicações conversacionais de propósito geral. Assistentes de programação, aplicações de geração aumentada por recuperação, agentes de manutenção e mecanismos de análise de código processam artefatos de software, executam múltiplas chamadas de inferência durante uma mesma tarefa e frequentemente manipulam código-fonte e documentação internos. Essas características tornam relevantes aspectos de engenharia como controle de acesso, rastreabilidade, compartilhamento de infraestrutura e governança sobre o uso dos modelos. Nesse contexto, a infraestrutura de inferência passa a integrar a própria arquitetura das ferramentas inteligentes de engenharia de software, aproximando este trabalho do domínio de *Intelligent Software Engineering*.

Este artigo apresenta uma plataforma multimodelo que centraliza o acesso a modelos internos e a provedores externos autorizados. A solução combina um *proxy* de inferência, servidores de modelos, bancos de dados, painel administrativo e mecanismos de monitoramento. Sua contribuição não é um novo algoritmo de inferência, mas uma arquitetura para compartilhamento, governança e rastreabilidade em um ambiente no qual diversas ferramentas inteligentes de engenharia de software utilizam a mesma infraestrutura.

Duas questões de pesquisa orientam o trabalho:

RQ1. Quais decisões de projeto sustentam uma plataforma multimodelo capaz de compartilhar uma capacidade de GPU

finita entre ferramentas inteligentes de engenharia de software, preservando controle de acesso, rastreabilidade e políticas de localização dos dados?

RQ2. Qual é o sobrecusto de latência introduzido pela plataforma em comparação ao acesso direto ao servidor de inferência?

As contribuições são: uma arquitetura organizada em planos de controle, dados e observabilidade; um modelo hierárquico de acesso; uma caracterização quantitativa preliminar da infraestrutura; um protocolo para avaliar o sobrecusto de latência; uma análise dos principais compromissos arquiteturais; e sete lições obtidas durante a implantação.

2 Contexto e Trabalhos Relacionados

O problema tratado neste trabalho situa-se na interseção entre plataformas de inferência para modelos de linguagem e engenharia de software para sistemas inteligentes. Embora esses sistemas atendam finalidades distintas, eles compartilham a necessidade de acessar modelos de linguagem e outros modelos de aprendizado de máquina de maneira controlada, auditável e eficiente.

Nos últimos anos, diversas soluções passaram a simplificar a operação desses modelos. Entre elas, o vLLM destaca-se como um servidor de inferência voltado ao atendimento eficiente de modelos generativos. Sua principal contribuição é otimizar o uso da memória da GPU por meio do mecanismo *PagedAttention*, permitindo maior aproveitamento do cache de atenção e maior paralelismo durante a geração de texto [6]. O vLLM tornou-se um dos servidores de inferência mais utilizados para modelos de pesos abertos e constitui a base de execução da plataforma apresentada neste trabalho. Entretanto, seu objetivo é executar modelos de forma eficiente, e não administrar organizações, projetos, usuários, permissões ou políticas de acesso compartilhadas entre diferentes aplicações.

Outra solução amplamente utilizada é o LiteLLM [12], que funciona como um proxy compatível com a API da OpenAI. O LiteLLM oferece uma interface unificada para diferentes provedores de modelos, abstraindo as diferenças entre APIs e permitindo autenticação por chaves virtuais, limitação de requisições, contabilização de uso e encaminhamento para múltiplos servidores. Essas funcionalidades reduzem o esforço de integração de aplicações clientes e justificam sua adoção na arquitetura proposta. Entretanto, o LiteLLM não fornece uma camada organizacional voltada ao gerenciamento de núcleos, projetos, inventário de infraestrutura, associação de requisições a equipes ou administração integrada de recursos computacionais. Neste trabalho, ele é utilizado como componente de roteamento, enquanto essas responsabilidades permanecem sob controle da plataforma.

O OpenRouter [5] também fornece uma interface compatível com a API da OpenAI e permite selecionar modelos de diversos provedores comerciais. Seu objetivo principal é facilitar o acesso a modelos hospedados por terceiros por meio de uma única API. Diferentemente da arquitetura proposta, o OpenRouter não opera modelos executados pela própria organização nem fornece mecanismos para integrar informações administrativas, infraestrutura local e políticas organizacionais de acesso. Consequentemente, ele não atende a cenários nos quais código-fonte ou documentos internos precisam permanecer em infraestrutura própria.

Outra linha de pesquisa concentra-se na utilização eficiente de GPUs compartilhadas. O Orca [7] propõe um mecanismo de escalonamento iterativo para aumentar a utilização dos aceleradores durante a geração de texto. O AlpaServe [8] explora a multiplexação estatística e o paralelismo de modelos para compartilhar um conjunto de GPUs entre diferentes modelos de aprendizado profundo. Esses trabalhos tratam principalmente do problema de alocação de recursos computacionais e da eficiência do processo de inferência. A plataforma apresentada neste artigo aproveita essas capacidades por meio do vLLM, mas atua em um nível arquitetural distinto. Seu objetivo não é definir como uma GPU deve ser utilizada internamente pelo servidor de inferência, mas como múltiplas ferramentas inteligentes compartilham modelos, credenciais e infraestrutura dentro de uma organização.

A literatura de MLOps também discute desafios relacionados à operação contínua de modelos de aprendizado de máquina. Sculley et al. [9] demonstraram que sistemas de aprendizado de máquina acumulam dívida técnica associada à implantação, manutenção e evolução da infraestrutura, enquanto Kreuzberger et al. [10] sistematizaram essas preocupações no contexto de Machine Learning Operations, propondo uma definição, uma arquitetura e princípios para MLOps. Essas contribuições tratam do ciclo de vida de modelos, incluindo treinamento, implantação, monitoramento e manutenção. O foco deste trabalho é mais específico: administrar uma plataforma compartilhada de inferência utilizada simultaneamente por diferentes ferramentas inteligentes de engenharia de software. Questões como controle de acesso, compartilhamento de GPUs, atribuição organizacional das requisições e coexistência de modelos internos e externos aparecem apenas marginalmente nessa literatura.

A Tabela 1 resume as principais diferenças entre as soluções discutidas e a plataforma proposta. O objetivo da comparação não é estabelecer superioridade entre ferramentas, mas evidenciar que cada uma foi concebida para responsabilidades distintas. Enquanto vLLM concentra-se na execução eficiente de modelos, LiteLLM atua como camada de abstração para diferentes provedores e OpenRouter simplifica o consumo de modelos hospedados externamente. A plataforma proposta combina essas capacidades e acrescenta mecanismos de organização administrativa, compartilhamento de infraestrutura, rastreabilidade e governança voltados especificamente ao suporte de ferramentas inteligentes utilizadas durante o desenvolvimento de software.

Tabela 1: Comparação funcional entre soluções relacionadas e a plataforma proposta.

Capacidade	vLLM	LiteLLM	OpenRouter	Plataforma
Serving interno	Sim	Integra	Não	Sim
Múltiplos provedores	Não	Sim	Sim	Sim
Compatibilidade OpenAI	Sim	Sim	Sim	Sim
Hierarquia organizacional	Não	Parcial	Não	Sim
Inventário de GPUs	Não	Não	Não	Sim
Associação a projetos	Não	Parcial	Não	Sim
Política interno/externo	Não	Configurável	Não	Sim
Monitoramento integrado	Não	Parcial	Não	Sim
Suporte a ferramentas de ES	Não	Não	Não	Sim

Em síntese, a plataforma proposta não substitui soluções como vLLM ou LiteLLM, mas as utiliza como componentes arquiteturais. Sua contribuição consiste em integrar esses mecanismos em uma infraestrutura comum para ferramentas inteligentes de engenharia de software, acrescentando governança, compartilhamento de recursos, rastreabilidade e controle organizacional que não aparecem de forma integrada nos trabalhos e plataformas analisados.

3 Requisitos e Restrições

A plataforma foi construída para um ambiente no qual várias ferramentas inteligentes compartilham modelos e GPUs. Três restrições orientaram o projeto.

A primeira é a localização dos dados. Aplicações que processam código proprietário ou documentos sob sigilo precisam utilizar modelos internos. Provedores externos podem ser cadastrados, mas seu uso depende das permissões da chave e da política do projeto. Portanto, impedir a saída dos dados é uma propriedade das regras de roteamento e autorização, não apenas da existência de servidores locais.

A segunda restrição é a capacidade finita e heterogênea. O ambiente possui três máquinas e doze GPUs. Quatro placas A100 de 80 GB executam o modelo de texto com paralelismo de tensor. Seis placas Quadro RTX 4000 de 8 GB atendem modelos de vetorização, reordenação e transcrição. Uma placa com 5 GB não comporta nenhum modelo do catálogo e uma A100 de 80 GB permanece disponível para outras cargas.

A terceira restrição é a pluralidade de sistemas. Assistentes de programação, agentes de manutenção, aplicações RAG e serviços de análise pertencem a projetos distintos, possuem responsáveis diferentes e geram perfis de carga próprios. A plataforma precisa autenticar cada chamada, autorizar o modelo solicitado, atribuir o consumo, permitir revogação e compartilhar os modelos sem criar uma cópia por aplicação.

4 Arquitetura da Plataforma

A arquitetura separa responsabilidades em três planos: controle, dados e observabilidade. Essa separação é lógica, pois alguns componentes participam de mais de um plano. O LiteLLM, por exemplo, atua no plano de controle ao receber configurações administrativas, no plano de dados ao autenticar e encaminhar requisições e no plano de observabilidade ao emitir registros de uso. A Figura 1 apresenta os principais componentes e fluxos.

4.1 Plano de Controle

O plano de controle mantém o cadastro de núcleos, projetos, usuários, chaves, modelos e máquinas. Um painel administrativo em React consulta uma API implementada em FastAPI, que persiste os dados no PostgreSQL da plataforma. A Figura 2 apresenta exemplos das telas utilizadas para administrar o catálogo de modelos, visualizar as máquinas e GPUs disponíveis e criar chaves de acesso vinculadas aos projetos.

O painel não atua apenas como interface de visualização. As operações realizadas por administradores e gerentes modificam o estado utilizado pelo plano de dados. Toda mutação é primeiro persistida no PostgreSQL da plataforma e, quando envolve modelos ou chaves, propagada ao LiteLLM por sua API administrativa.

Essa organização distribui responsabilidades entre os componentes. O PostgreSQL da plataforma mantém a estrutura organizacional, incluindo núcleos, projetos, usuários, máquinas e os vínculos entre essas entidades. O LiteLLM mantém o estado utilizado durante a autenticação e a autorização das requisições.

A separação permite reutilizar os mecanismos nativos do proxy, mas cria estado distribuído. Uma operação pode ser concluída em um componente e falhar antes de alcançar o outro. Por isso, a ordem das operações, a idempotência e os procedimentos de reconciliação fazem parte da arquitetura, e não apenas da implementação.

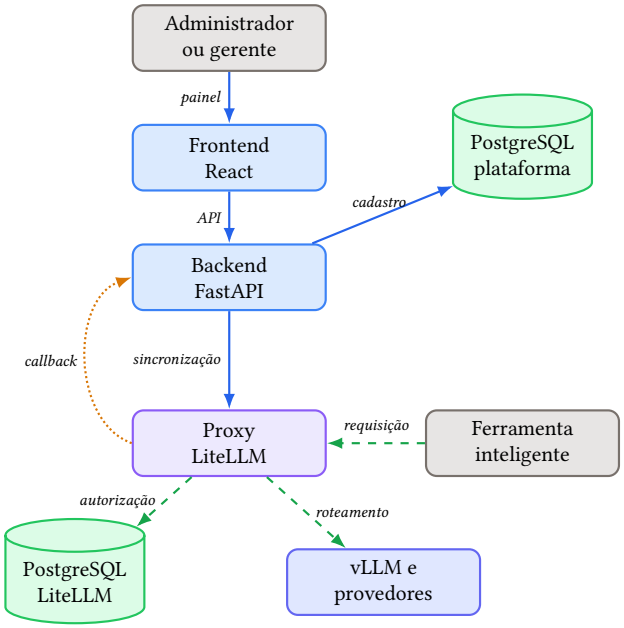

Figura 1: Arquitetura da plataforma e separação dos fluxos nos planos de controle, dados e observabilidade.

O acesso administrativo possui dois papéis. Administradores podem operar todos os núcleos e projetos cadastrados. Gerentes possuem acesso restrito ao núcleo ao qual estão vinculados. Essa separação permite delegar a gestão de projetos sem conceder acesso global à plataforma.

4.2 Plano de Dados

O plano de dados conduz as requisições das ferramentas inteligentes até os modelos autorizados. Uma aplicação compatível com a API da OpenAI envia a chamada à plataforma e apresenta uma chave de acesso. O LiteLLM autentica a credencial, verifica se o modelo solicitado está autorizado para aquela chave e encaminha a requisição ao servidor correspondente.

Os modelos internos são executados em processos vLLM isolados em contêineres. Quando permitido pela configuração do projeto, uma chamada também pode ser encaminhada a um provedor externo. Para a ferramenta cliente, os diferentes destinos são apresentados por uma interface comum.

Essa abstração beneficia diretamente ferramentas inteligentes de engenharia de software. Assistentes de programação, agentes de manutenção, aplicações RAG, serviços de análise de código e mecanismos de revisão podem utilizar a mesma interface, mesmo quando dependem de modelos diferentes. A infraestrutura pode alterar internamente a máquina, o servidor ou o provedor sem exigir modificações no contrato consumido por essas ferramentas.

A adoção de uma interface compatível com a API da OpenAI também reduz o esforço de integração. Muitas bibliotecas e ferramentas já implementam esse contrato, o que permite substituir principalmente o endereço do serviço e a chave de autenticação. A organização evita, assim, manter adaptações específicas para cada cliente.

Entretanto, a compatibilidade com uma API amplamente adotada não constitui uma garantia permanente. Durante a operação da

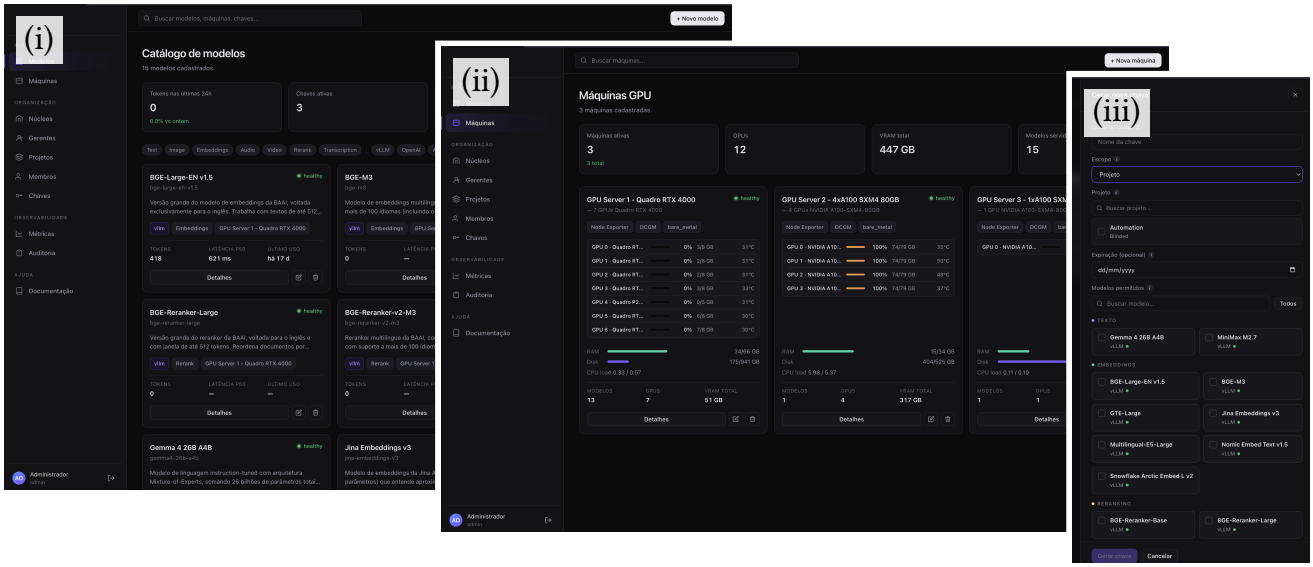

Figura 2: Exemplos de telas do painel de administração: (i) catálogo de modelos, (ii) máquinas e GPUs e (iii) criação de uma chave de acesso.

plataforma, uma atualização do LiteLLM modificou o tratamento de um endpoint e produziu incompatibilidade com um servidor de inferência já implantado. A mitigação consistiu em fixar uma versão anterior do proxy. O incidente demonstrou que o contrato deve ser tratado como dependência versionada e submetido a testes antes de cada atualização.

4.3 Plano de Observabilidade

O plano de observabilidade registra o resultado das chamadas e coleta informações sobre as máquinas. Após atender uma requisição, o LiteLLM envia ao backend um callback contendo o modelo utilizado, a quantidade de tokens, a latência e uma identificação derivada da chave de acesso.

O backend confirma o recebimento e persiste o evento de forma assíncrona, evitando bloquear o processamento do proxy. A partir da identificação da chave, a plataforma associa a chamada ao projeto e, quando aplicável, ao usuário responsável.

Cada máquina também executa dois exportadores Prometheus. O Node Exporter fornece métricas de CPU, memória, disco e rede. O DCGM Exporter fornece utilização, memória ocupada e temperatura das GPUs NVIDIA. O backend consulta periodicamente esses endpoints e mantém uma visão consolidada das condições da infraestrutura.

Esses registros apoiam auditoria, diagnóstico de falhas e planejamento de capacidade. Também permitem identificar quais ferramentas e projetos geram maior consumo, quais modelos apresentam maior latência e quais GPUs operam próximas do limite.

Na implantação inicial, o registro inclui o texto da última mensagem do usuário. Essa informação facilita a investigação de incidentes, mas cria um repositório central de código e conteúdo potencialmente sensível. Uma implantação de produção deve aplicar políticas de minimização, mascaramento, criptografia, retenção e controle de acesso. Quando o texto integral não for indispensável, a plataforma deve preferir metadados ou representações reduzidas.

4.4 Hierarquia de Acesso e Atribuição

O acesso é organizado em quatro níveis: núcleo, projeto, usuário e chave. Um núcleo agrupa projetos, e cada projeto reúne sistemas, usuários e credenciais. Uma chave pode representar uma aplicação inteira ou estar vinculada a um usuário específico.

Seja M o catálogo de modelos disponíveis e $A(k)$ o subconjunto de modelos autorizado para uma chave k. Uma requisição só pode ser encaminhada quando o modelo solicitado pertence a $A(k)$.

Chaves de projetos diferentes podem acessar a mesma instância de um modelo, desde que possuam autorização. O compartilhamento ocorre no servidor de inferência, sem criação de uma cópia do modelo por projeto, usuário ou aplicação. Essa decisão reduz a replicação de modelos e permite repartir a mesma capacidade de GPU entre ferramentas distintas.

O PostgreSQL da plataforma não armazena a chave em texto legível. Ele mantém seu hash, estado, data de expiração e identificador no LiteLLM. Quando o callback é recebido, o hash permite localizar o projeto e o usuário associados à chamada.

Essa estratégia reduz a exposição direta das credenciais, mas não elimina todos os riscos. Uma chave comprometida ainda pode ser utilizada até sua expiração ou revogação. Além disso, quando várias pessoas compartilham a mesma chave de aplicação, a atribuição alcança o nível do projeto ou sistema, mas não identifica com precisão o operador humano.

4.5 Sincronização e Revogação

O estado de uma chave fica dividido entre dois componentes. O PostgreSQL da plataforma conhece sua relação com núcleos, projetos e usuários. O LiteLLM mantém as permissões utilizadas durante a autorização.

Nenhum dos dois componentes possui, isoladamente, o estado completo. Por isso, leituras administrativas podem exigir consulta ao LiteLLM, especialmente para recuperar os modelos autorizados para cada chave.

A revogação é aplicada primeiro no LiteLLM e depois no PostgreSQL da plataforma. Essa ordem prioriza o bloqueio efetivo da credencial. Se ocorrer uma falha intermediária, a chave pode permanecer temporariamente apresentada como ativa no painel, mas já não será aceita pelo proxy.

A criação e a atualização também exigem tratamento de falhas parciais. Operações repetidas devem utilizar identificadores estáveis e produzir o mesmo resultado sempre que possível. Em inicializações e recuperações, a plataforma executa procedimentos de reconciliação para identificar registros ausentes ou divergentes.

A implementação atual ainda não oferece uma transação distribuída entre os dois bancos. Portanto, existe uma janela de inconsistência eventual. Essa limitação motiva a adoção futura de filas persistentes, registros de operações pendentes e reconciliação bidirecional.

4.6 Decisões e Compromissos Arquiteturais

As principais decisões foram orientadas por interoperabilidade, controle organizacional, compartilhamento de recursos e rastreabilidade. Cada decisão, contudo, introduz um compromisso, resumido na Tabela 2.

Tabela 2: Decisões de projeto e compromissos arquiteturais.

Decisão	Benefício	Compromisso
API compatível	Integração com ferramentas existentes	Dependência de contratos e versões externas
Proxy central	Políticas e roteamento uniformes	Ponto comum de falha e contenção
Modelo compartilhado	Evita cópias por aplicação	Exige limites, filas e isolamento
Estado dividido	Reutiliza autorização do LiteLLM	Pode produzir divergências
Registro por chave	Atribuição por projeto e usuário	Chaves compartilhadas reduzem granularidade
Logs de prompts	Facilitam auditoria e diagnóstico	Aumentam o risco de exposição
Modelos internos e externos	Flexibilidade de capacidade e custo	Exige política explícita de localização

A adoção da API compatível reduz o esforço de integração, mas submete a plataforma à evolução de contratos externos. O proxy centraliza políticas, mas também concentra falhas. O compartilhamento reduz cópias, mas cria contenção entre aplicações. O estado dividido permite reutilizar o LiteLLM, mas exige reconciliação. A observabilidade amplia a rastreabilidade, mas precisa ser equilibrada com a proteção dos dados registrados.

Resposta à RQ1. A plataforma é sustentada por cinco decisões principais: (i) adoção de uma interface compatível com ferramentas existentes; (ii) separação dos fluxos em planos de controle, dados e observabilidade; (iii) organização do acesso em núcleos, projetos, usuários e chaves; (iv) atribuição de cada requisição por meio da identificação da chave; e (v) compartilhamento de uma única instância de cada modelo entre aplicações autorizadas.

5 Avaliação Preliminar

A avaliação contém uma caracterização quantitativa da implantação e um experimento preliminar de latência.

5.1 Caracterização da Infraestrutura

A implantação utiliza três máquinas e doze GPUs. Dos 453 GB de memória instalada, 448 GB são utilizáveis pelos modelos atuais. Aproximadamente 1,1% da memória está associada à GPU de 5 GB que não comporta nenhum modelo do catálogo. O resultado mostra que capacidade instalada não equivale a capacidade utilizável.

As quatro A100 utilizadas pelo modelo de texto fornecem 320 GB agregados. Os modelos auxiliares utilizam um conjunto separado de seis GPUs, com 48 GB agregados. Essa divisão evita que serviços de vetorização, reordenação e transcrição disputem diretamente a memória reservada pelo modelo de geração.

A plataforma mantém uma instância de cada modelo disponível para todas as chaves autorizadas. O benefício máximo de consolidação aumenta com a quantidade de aplicações que, sem a plataforma, manteriam cópias independentes. Como a configuração anterior de cada aplicação não foi instrumentada, o estudo não quantifica GPUs ou memória efetivamente economizadas.

5.2 Experimento de Latência

O experimento E1 compara o acesso direto ao servidor vLLM com o acesso intermediado pelo LiteLLM. As duas condições utilizam o mesmo modelo, servidor, parâmetros e rede.

O corpus contém dez prompts sintéticos relacionados a geração e teste de código, revisão de segurança, análise de complexidade, RAG, diagnóstico de falhas e comparação de arquiteturas. Os prompts possuem entre 142 e 162 caracteres e a resposta é limitada a 256 tokens. A concorrência é igual a um.

Cada execução completa contém os dez prompts nas duas condições. A unidade de análise é a execução, pois requisições realizadas no mesmo período compartilham temperatura da GPU, tráfego, estado do servidor e possíveis efeitos de cache.

Para cada execução são calculadas a mediana e o percentil 95 do tempo até o primeiro token e da latência fim a fim. A análise deve utilizar diferenças pareadas entre as condições. Intervalos de confiança podem ser estimados por bootstrap BCa [11].

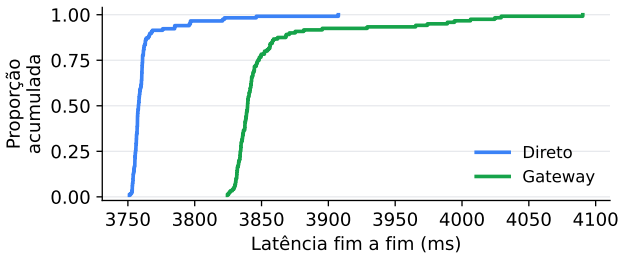

Figura 3: Distribuição acumulada da latência fim a fim.

A Figura 3 apresenta a distribuição acumulada da latência fim a fim para as duas abordagens. Observa-se que a curva correspondente à plataforma está deslocada para valores mais elevados, indicando maior latência em relação ao acesso direto. A mediana da latência fim a fim aumentou de 3757,7 ms para 3839,7 ms, correspondendo a um sobrecusto mediano de 82,2 ms. O intervalo de confiança BCa de 95% para esse sobrecusto foi de 77,7 ms a 86,4 ms.

Resposta à RQ2. A plataforma introduziu um sobrecusto mediano de 52,0 ms no TTFT, com intervalo de confiança BCa de 95% entre 49,2 ms e 56,0 ms. Na latência fim a fim, o sobrecusto mediano foi de 82,2 ms, com intervalo de confiança BCa de 95% entre 77,7 ms e 86,4 ms. Portanto, a camada adicional de autenticação, autorização, roteamento e observabilidade aumentou de forma consistente a latência das requisições em relação ao acesso direto ao servidor de inferência.

6 Lições Aprendidas

A implantação e a operação da plataforma permitiram identificar aspectos que influenciaram diretamente seu projeto e sua evolução. As lições a seguir sintetizam decisões tomadas durante o desenvolvimento, problemas encontrados e seus respectivos impactos sobre a arquitetura.

- **L1.** Atualizações de componentes mantidos por terceiros podem alterar o encaminhamento das requisições e comprometer a compatibilidade com servidores ou aplicações já implantados. Durante a operação, uma atualização do LiteLLM exigiu a fixação de uma versão anterior, mostrando que APIs e proxies devem ser tratados como dependências versionadas e submetidos a testes de regressão.
- **L2.** Quando diferentes componentes armazenam partes de um mesmo estado, suas responsabilidades devem ser claramente definidas. Na plataforma, o PostgreSQL mantém a estrutura organizacional, enquanto o LiteLLM controla as permissões de inferência. Essa separação exige sincronização e reconciliação entre os componentes.
- **L3.** A ordem das operações influencia a segurança do sistema. A revogação de uma chave ocorre primeiro no LiteLLM e depois no banco da plataforma, evitando que uma falha intermediária mantenha a credencial ativa.
- **L4.** O planejamento da infraestrutura deve considerar a capacidade efetivamente utilizável das GPUs. Durante a implantação, uma GPU de 5 GB não comportou nenhum modelo do catálogo, apesar de integrar o parque computacional.
- **L5.** Limites de concorrência devem ser conhecidos pelas aplicações clientes. O LiteLLM rejeita requisições acima do limite com HTTP 429, mas aplicações que desconhecem essa política podem tratá-las como falhas inesperadas. Documentação e mecanismos de repetição tornam a integração mais robusta.
- **L6.** A contabilização do uso deve ser persistente e idempotente. Estruturas temporárias em memória podem perder ou duplicar informações após reinicializações, prejudicando a auditoria e a consolidação do consumo por projeto.

7 Ameaças à Validade

Na validade externa, o estudo considera uma única organização, um conjunto específico de modelos e uma infraestrutura com doze GPUs. Os resultados podem não se repetir em outros ambientes. O corpus contém apenas dez prompts curtos e não representa agentes com contextos longos ou várias chamadas encadeadas.

Na validade interna, tráfego de rede, temperatura, cache e processos concorrentes podem afetar a latência. O uso da mesma rede, do mesmo servidor e de condições pareadas reduz parte desses efeitos. A ordem das condições deve ser aleatorizada em execuções futuras.

Na validade de construto, TTFT e latência fim a fim não capturam isoladamente throughput, custo, fairness ou qualidade das respostas. A saída fixa em 256 tokens reduz variabilidade, mas também limita a representatividade.

A contribuição arquitetural também possui limitações. A plataforma integra componentes existentes e não propõe um novo algoritmo de serving. O ganho de consolidação não foi medido em GPUs, memória ou custo. Também não foram avaliados throughput sob concorrência, isolamento entre projetos, disponibilidade do proxy e comportamento durante falhas.

8 Considerações Finais

Este artigo apresentou uma plataforma multimodelo para apoiar ferramentas inteligentes de engenharia de software em ambientes organizacionais. A arquitetura integra modelos executados internamente e provedores externos autorizados sob uma interface comum, centraliza o controle de acesso, compartilha a capacidade de GPU entre diferentes aplicações e mantém a atribuição das requisições aos respectivos projetos e usuários. A avaliação mostrou que essa camada introduz um sobrecusto mediano de 52,0 ms no TTFT e de 82,2 ms na latência fim a fim, enquanto as lições aprendidas evidenciaram desafios relacionados à compatibilidade entre componentes, sincronização de estado, revogação de credenciais, contabilização de uso e escalonamento horizontal.

Como trabalhos futuros, pretende-se ampliar a avaliação para diferentes níveis de concorrência, tamanhos de contexto, modelos, provedores e configurações de hardware, utilizando cargas de trabalho representativas do uso real das ferramentas. Também serão investigadas métricas como throughput em tokens por segundo, tempo em fila, utilização de GPU, taxa de respostas HTTP 429, consumo por projeto e ganho de capacidade decorrente da consolidação. Além disso, a plataforma deverá evoluir com mecanismos de reconciliação persistente entre componentes, cache compartilhado entre réplicas do proxy, políticas de retenção e proteção dos prompts registrados e estratégias para reduzir o impacto de falhas na camada central de inferência.

Disponibilidade de Artefatos

Os artefatos disponibilizados para apoiar a compreensão e a replicação deste estudo estão disponíveis publicamente no repositório.[1]

Agradecimentos

Este trabalho foi parcialmente financiado pelo projeto “ISOP Engineering: Métodos e Ferramentas de Engenharia Inteligente para Sensoriamento Inteligente”, apoiado pelo CENTRO DE COMPETÊNCIA EMBRAPII VIRTUS EM HARDWARE INTELIGENTE PARA INDÚSTRIA – VIRTUS-CC, com recursos financeiros do PPI HardwareBR do MCTI, conforme o termo nº 055/2023, firmado com a EMBRAPII.

Referências

[1] Peng, S. *et al.* The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. *arXiv preprint arXiv:2302.06590*, 2023.
[2] Jimenez, C. E. *et al.* SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In: *International Conference on Learning Representations (ICLR)*, p. 54107–54157, 2024.
[3] Yang, J. *et al.* SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In: *Advances in Neural Information Processing Systems (NeurIPS)*, v. 37, p. 50528–50652, 2024.
[4] Anthropic. Is my data used for model training? Disponível em: https://privacy.claude.com/en/articles/7996868-is-my-data-used-for-model-training.
[5] OpenRouter. Privacy Policy. Disponível em: https://openrouter.ai/privacy.
[6] Kwon, W. *et al.* Efficient Memory Management for Large Language Model Serving with PagedAttention. In: *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, p. 611–626, 2023.
[7] Yu, G. *et al.* Orca: A Distributed Serving System for Transformer-Based Generative Models. In: *16th USENIX Symposium on Operating Systems Design and Implementation*, p. 521–538, 2022.
[8] Li, Z. *et al.* AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, p. 663–679, 2023.
[9] Sculley, D. *et al.* Hidden Technical Debt in Machine Learning Systems. In: *Advances in Neural Information Processing Systems (NeurIPS)*, v. 28, 2015.
[10] Kreuzberger, D.; Kühl, N.; Hirschl, S. Machine Learning Operations: Overview, Definition, and Architecture. *IEEE Access*, v. 11, p. 31866–31879, 2023.
[11] Efron, B.; Tibshirani, R. J. *An Introduction to the Bootstrap.* Chapman & Hall, 1993.
[12] BerriAI. LiteLLM: Call LLM APIs using the OpenAI format. Disponível em: https://github.com/BerriAI/litellm.

[1] https://github.com/platTool-cmyk/supplementary-material