Documentação - Decisões Técnicas

Projeto IsCoolGPT

BACKEND

O desenvolvimento do backend do IsCoolGPT começou pela definição da linguagem e do framework que seriam utilizados. Optou-se por Python 3.11 devido ao conjunto de melhorias introduzidas nessa versão, especialmente no tratamento de erros e no desempenho geral, que apresentou ganhos em relação às versões anteriores. Python é também uma linguagem amplamente utilizada em aplicações educacionais, de prototipagem rápida e em sistemas que necessitam de integração com APIs externas, como é o caso do serviço de inteligência artificial que alimenta o IsCoolGPT. A escolha do Flask foi fruto de uma análise comparativa entre frameworks Python populares, onde foram considerados aspectos como simplicidade, curva de aprendizado, compatibilidade com estruturas modulares e facilidade de containerização. O FastAPI chegou a ser avaliado por sua performance superior e tipagem automática, mas o Flask demonstrou maior adequação ao escopo do projeto, que priorizava transparência, flexibilidade e familiaridade.

A arquitetura do backend foi organizada utilizando Blueprints, garantindo separação clara entre funcionalidades, rotas e camadas de lógica. Essa opção traz vantagens de manutenção, permitindo que novos módulos sejam adicionados com impacto mínimo no restante do sistema. A modularidade proporcionou maior clareza durante o desenvolvimento e favoreceu o diagnóstico de erros, já que cada área funcional possui responsabilidade específica dentro da estrutura do projeto. O sistema de logging, desenvolvido para registrar acessos, erros e eventos internos, foi projetado com atenção à futura necessidade de diagnósticos em produção. A integração com o serviço Google Gemini foi tratada como um elemento sensível devido à latência variável de modelos de linguagem, o que influenciou também a definição do fluxo de requisições e a necessidade de observabilidade.

A containerização do backend com Docker foi essencial para garantir portabilidade e consistência entre ambientes. A construção da imagem utilizou práticas que minimizam tamanho e aumentam eficiência, como a criação de um arquivo .dockerignore para remover arquivos irrelevantes, a instalação com pip sem cache e a definição de dependências estritamente necessárias. Essas otimizações resultaram em uma imagem significativamente menor, impactando positivamente no tempo de build e na velocidade de deploy no ambiente gerenciado da AWS. O Dockerfile foi estruturado para expor a porta correta utilizada pelo Flask, garantir replicabilidade do ambiente e

permitir integração nativa com ECS Fargate, que requer uso de imagens leves e de execução rápida. Tanto no ambiente local quanto na nuvem, a aplicação se comportou de forma consistente graças ao isolamento provido pela containerização.

Durante os testes iniciais, o backend apresentou alguns comportamentos instáveis decorrentes da carga imposta por requisições simultâneas ao modelo externo. A análise de consumo de CPU e memória demonstrou necessidade de ajustar configurações internas, como o número de workers do servidor de aplicação e a alocação de memória. Esses diagnósticos foram feitos observando logs e métricas de runtime, reforçando o papel dos mecanismos de observabilidade integrados ao backend. Uma vez ajustadas essas configurações, o sistema atingiu o equilíbrio necessário entre desempenho e estabilidade, apresentando respostas consistentes e processando requisições sem degradação perceptível em cenários de teste com múltiplos usuários simultâneos.

O backend, portanto, foi desenvolvido com foco em simplicidade, estabilidade e compatibilidade com os ambientes da AWS. A combinação de Python, Flask, modularização via Blueprints, containerização bem construída e integração com o Gemini formou uma base sólida que permitiu ao projeto evoluir com segurança e previsibilidade, atendendo aos requisitos de uma aplicação cloud-native educacional e escalável.

CI/CD

O pipeline de integração e entrega contínua do projeto foi implementado com GitHub Actions, uma escolha que refletiu a necessidade de uma solução integrada diretamente ao repositório de código, de fácil configuração e que permitisse automações completas sem dependências externas. A decisão de utilizar GitHub Actions foi reforçada pela disponibilidade de actions prontas para uso, inclusive para autenticação com serviços da AWS, além da vantagem de ser uma solução gratuita para repositórios públicos, o que se encaixava no contexto acadêmico do projeto. O workflow desenvolvido englobou desde o checkout do código até o build da imagem Docker, push para o Amazon ECR e por fim a atualização do serviço no ECS Fargate, tudo de maneira automática sempre que houvesse commits enviados para as branches configuradas.

A configuração do pipeline foi estruturada de forma a interoperar com o repositório ECR do projeto e a provocar deployments automáticos no ECS. A automação incluiu uso de actions oficiais da AWS para configurar credenciais temporárias, validar autenticações e realizar login no registro de container antes de realizar operações de push. Em seguida, o pipeline executava a construção da imagem, marcando-a com o hash do commit e também atualizando a tag latest, o que facilita identificar versões em produção ou associar falhas a versões específicas. Após o envio da imagem ao ECR,

uma chamada ao AWS CLI realizava a atualização do serviço ECS, forçando um novo deployment com a imagem atualizada.

Para manter a segurança do pipeline, todos os segredos sensíveis necessários ao processo foram armazenados exclusivamente no GitHub Secrets. Isso garantiu que credenciais como chaves de acesso da AWS e a chave da API do Gemini nunca fossem expostas em logs ou arquivos versionados. Durante o desenvolvimento, foi inicialmente considerada a configuração dessas credenciais diretamente via pipeline para dentro da Task Definition, mas essa estratégia foi descartada ao se perceber que isso poderia expor informações críticas. A chave da API foi então mantida diretamente na definição de tarefa na AWS, prática aceitável dentro do contexto acadêmico, mas que seria reformulada para uso do Secrets Manager em um ambiente de produção real.

O controle de versionamento adotou um fluxo simplificado inspirado no Git Flow, com divisão funcional entre ambientes de desenvolvimento, homologação e produção. Branches feature foram utilizadas para novas funcionalidades, staging serviu como ambiente intermediário para testes de aprovação e a main representou o código estável e pronto para deployment definitivo. Isso permitiu que o pipeline CI/CD realizasse deploys distintos conforme a branch de origem, dando previsibilidade ao ciclo de desenvolvimento e facilitando rollback caso erros fossem identificados. Esse modelo contribuiu para a organização do trabalho e para a confiabilidade das entregas.

O pipeline enfrentou desafios técnicos durante o desenvolvimento, especialmente relacionados a conflitos de merge que acabaram corrompendo a sintaxe YAML. Esses problemas levaram à interrupção temporária de deploys automáticos e mostraram a importância de práticas adequadas de revisão e limpeza de código. Após correções, o workflow voltou a operar com estabilidade e se tornou um dos pilares da automação do projeto. Ao final, o GitHub Actions se mostrou uma ferramenta eficiente, previsível e compatível com o escopo da aplicação, permitindo que o processo de entrega fosse totalmente automatizado e integrado ao fluxo de trabalho.

AWS

A infraestrutura em nuvem do projeto foi projetada inteiramente sobre os serviços da AWS, com foco em escalabilidade, desempenho, segurança e simplicidade operacional. A decisão mais importante nesse contexto foi a de utilizar o Amazon ECS com Fargate como plataforma de execução dos containers. Essa escolha foi feita após análise detalhada de alternativas como EC2 puro, ECS com EC2 Launch Type, Amazon EKS, AWS Lambda e Elastic Beanstalk. O EC2 foi descartado por exigir manutenção completa dos servidores, desde atualizações até escalonamento manual. O ECS com EC2 Launch Type, embora mais econômico, demandaria o gerenciamento de instâncias, o que aumentaria a complexidade de forma desnecessária. O EKS foi

considerado poderoso mas excessivamente complexo e oneroso para um projeto com escopo acadêmico. Lambda apresentou limitações de tempo de execução e riscos de cold start, incompatíveis com uma API que depende de requisições potencialmente longas ao modelo Gemini. Por fim, o Elastic Beanstalk foi rejeitado por esconder detalhes da infraestrutura que precisavam ser explicitamente explorados no contexto deste projeto.

Com isso, o ECS Fargate se mostrou a melhor solução. Ele permitiu a execução dos containers de forma serverless, eliminando a necessidade de gerenciar servidores e oferecendo isolamento de kernel para cada task. A capacidade de cobrança granular por uso de CPU e memória também ajudou a manter os custos dentro de limites adequados. A definição da task foi feita com 1 vCPU e 3GB de memória, valores escolhidos após extensos testes que apontaram gargalos quando configurações menores foram utilizadas. Estiveram entre os problemas mitigados o aumento severo da latência, desaceleração do processamento durante picos e acionamento do OOM Killer em configurações inferiores. O health check configurado garantiu que tasks com falhas internas fossem substituídas automaticamente, evitando situações em que o container estivesse ativo, porém funcionalmente indisponível.

A criação de uma VPC customizada foi outra decisão estratégica essencial. Essa escolha ofereceu controle total sobre as subnets, tabelas de roteamento, gateways e regras de segurança. Duas subnets públicas foram distribuídas entre duas Availability Zones diferentes, contribuindo para alta disponibilidade. Subnets privadas também foram criadas para futuros serviços, ainda que não utilizadas no presente estágio. O Internet Gateway, associado às subnets públicas, garantiu conectividade, enquanto as tabelas de rota foram configuradas para permitir tráfego externo e interno conforme necessário.

Um dos maiores desafios técnicos ocorreu justamente após a migração da aplicação para a VPC customizada. Todas as requisições externas passaram a falhar com timeout, mesmo com o container executando corretamente. A investigação aprofundada revelou que o Security Group padrão da VPC não possuía regras permitindo tráfego externo na porta utilizada pela aplicação. Somente tráfego interno era permitido por uma regra de self-reference. A correção envolveu adicionar explicitamente a permissão para entrada de tráfego TCP na porta 8080 a partir de qualquer origem. Após a correção, a aplicação voltou a responder normalmente. Essa experiência demonstrou a importância crítica da compreensão dos mecanismos de segurança de rede da AWS.

O controle de permissões via IAM também passou por ajustes devido à descoberta de que a Task Role estava com permissões excessivamente amplas durante a primeira

configuração. A separação entre Task Execution Role e Task Role foi aplicada conforme boas práticas da AWS, reduzindo a superfície de ataque e mitigando riscos de exploração em caso de falhas mais severas ou vulnerabilidades internas. Todo esse trabalho contribuiu para uma infraestrutura mais segura e alinhada ao princípio de menor privilégio.

O Amazon ECR foi utilizado como registro privado de imagens Docker. Essa escolha garantiu segurança, integração nativa com ECS, scan de vulnerabilidades automático e políticas de ciclo de vida para gerenciamento de versões antigas. Isso trouxe mais controle sobre armazenamento de imagens e facilitou a manutenção do ambiente durante todo o ciclo do projeto. A observabilidade foi construída com CloudWatch Logs e Container Insights, que forneceram métricas vitais para ajustes de desempenho, detecção de gargalos e verificações de estabilidade do sistema.

CONCLUSÃO

A estratégia técnica adotada no projeto IsCoolGPT demonstrou equilíbrio entre modernidade, segurança, praticidade e alinhamento às melhores práticas de engenharia de software. O backend construído em Python e Flask entregou simplicidade, clareza arquitetural e facilidade de extensão. A containerização trouxe reprodutibilidade e portabilidade, enquanto o pipeline de CI/CD automatizou todo o ciclo de entrega com eficiência e precisão. A infraestrutura baseada em AWS, com destaque para ECS Fargate, permitiu operar com isolamento, escalabilidade e robustez, mantendo o controle necessário para fins acadêmicos e possibilitando experimentação em cenários reais de produção. Os desafios enfrentados ao longo do caminho proporcionaram aprendizados importantes sobre segurança de rede, observabilidade e gestão de recursos em ambientes cloud-native. No conjunto, as decisões tomadas resultaram em uma arquitetura coerente, estável e demonstrativa de habilidades essenciais para o desenvolvimento de sistemas modernos, reforçando a capacidade do projeto de servir como base sólida para evoluções futuras ou para aplicações similares em contextos de maior escala.