Cloud AutoDroid: Um Sistema Distribuído Escalável para Execução de Ferramentas de IA Generativa

Luiz Felipe Laviola, Diego Kreutz, Rodrigo Brandão Mansilha

[1] Laboratório de Estudos Avançados em Computação (LEA)
Programa de Pós-Graduação em Engenharia de Software (PPGES)
Universidade Federal do Pampa (UNIPAMPA)

luiz@laviola.dev, diegokreutz@unipampa.edu.br, mansilha@unipampa.edu.br

Resumo. *A Cloud AutoDroid foi desenvolvida para resolver problemas de escalabilidade na execução de experimentos com redes neurais complexas, como a MalSynGen, que exigem alto poder computacional. Com uma arquitetura distribuída, a ferramenta permite a execução autoescalável de tarefas de IA. Disponível como um serviço SaaS (Software as a Service), a Cloud AutoDroid oferece uma plataforma para experimentação em larga escala, embora ainda requeira intervenção para a alocação inicial de nós.*

Abstract. *The Cloud AutoDroid was developed to solve scalability problems in the execution of experiments with complex neural networks, such as MalSynGen, which demand high computational power. With a distributed architecture, the tool allows for the autoscaling execution of AI tasks. Available as a SaaS (Software as a Service), the Cloud AutoDroid offers a platform for large-scale experimentation, although it still requires intervention for the initial allocation of nodes.*

1. Introdução

A utilização de ferramentas como a MalSynGen [Nogueira et al. 2024a] para geração de dados sintéticos demanda um poder computacional significativo, frequentemente indisponível para o usuário. Por exemplo, a identificação dos melhores hiperparâmetros para gerar dados sintéticos de qualidade para detecção de malware Android pode exigir mais de 20 combinações de hiperparâmetros por conjunto de dados [Nogueira et al. 2024b]. Esse processo deve ser repetido para cada conjunto de dados, o que pode se tornar proibitivo em termos de tempo e recursos. Em testes recentes, foram necessárias mais de 480 execuções da MalSynGen para otimizar hiperparâmetros em 10 conjuntos de dados distintos [Nogueira et al. 2024b].

O tempo de execução da rede neural da MalSynGen depende diretamente do tamanho do conjunto de dados de entrada. Por exemplo, um conjunto de dados como o MH-100K [Bragança et al. 2023] pode levar mais de 10 horas por execução. Considerando 480 execuções de 10 horas cada, o tempo total estimado seria de 200 dias em uma única máquina. Esse cenário evidencia um problema crítico de escalabilidade em experimentos com ferramentas como a MalSynGen.

Para além disso, no projeto Malware DataLab o objetivo é disponibilizar uma plataforma online de aprendizado e experimentação para a MalSynGen. O objetivo é atender dezenas a centenas de usuários para a execução de experimentos

reais com a MalSynGen. Ou seja, o problema de escala fica muitas vezes mais amplificado. Portanto, fica evidente a necessidade de uma plataforma que permita a execução autoescalável de experimentos utilizando redes neurais complexas como as da MalSynGen.

A Cloud AutoDroid [Laviola et al. 2023, Laviola et al. 2024] nasceu para atender essa demanda de execução auto-escalável de experimentos utilizando redes neurais complexas como as da MalSynGen. A primeira versão da AutoDroid [Laviola et al. 2023] era limitada a facilitar a execução local de ferramentas como a MalSynGen, ou seja, evitava processos e procedimentos manuais necessários numa instalação local ou na utilização direta de docker. Já a segunda versão, que derivou de um processo de engenharia de software [Laviola et al. 2024], contempla a utilização de uma arquitetura distribuída para permitir a execução em escala de ferramentas de IA como a MalSynGen.

A evolução da AutoDroid para uma arquitetura distribuída seguiu princípios de engenharia de software, incluindo a definição de requisitos, design da arquitetura, implementação modular e testes rigorosos. A arquitetura distribuída foi projetada para permitir a escalabilidade horizontal, ou seja, a adição de novos nós para aumentar a capacidade de processamento conforme a demanda. A implementação modular facilitou a manutenção e a evolução do sistema, permitindo a adição de novas funcionalidades e a integração com outras ferramentas de IA. Os testes garantiram a qualidade e a confiabilidade do sistema, assegurando que ele pudesse lidar com a complexidade dos experimentos de geração de dados sintéticos.

Neste trabalho apresentamos a primeira versão da Cloud AutoDroid como um serviço, ou seja *Software as a Service* (SaaS). Nessa primeira versão SaaS, a alocação de nós ainda exige a intervenção do administrativo para realizar o *setup* dos nós, ou seja, inicializar um novo nó com o software básico do trabalhador (aqui denominado *worker*) que irá integrar a arquitetura distribuída e escalável para execução de tarefas complexas de aplicações que envolvem IA, como a MalSynGen.

2. Trabalhos Relacionados

O problema de executar *workflows* de IA em escala é um desafio que vem sendo cada vez mais investigado na literatura [Wozniak et al. 2018, Zaharia et al. 2018, Chen et al. 2021, Krawczuk et al. 2021, Bhatia et al. 2021, Tagliabue et al. 2023, Polyxronou 2024]. Entretanto, a maioria das iniciativas, como o MLflow[1] [Zaharia et al. 2018] e o MetaFlow [Tagliabue et al. 2023], concentram-se em fornecer *frameworks* para projetos de aprendizado de máquina (ML) que visam aumentar a produtividade dos profissionais de dados, abstraindo a execução do código de ML da definição da lógica de negócios. Essas soluções são amplamente adotadas para gerenciar o ciclo de vida de modelos de ML, desde o desenvolvimento até a implantação, mas não são projetadas para lidar com aplicações de domínio específico que demandam alta escalabilidade e execução autônoma.

Por outro lado, soluções como a MuMMI [Chen et al. 2021] e suas evoluções (e.g., [Bhatia et al. 2021]) apresentam *frameworks* escaláveis e generalizáveis

[1] `https://mlflow.org`

que combinam modelos usando aprendizado de máquina e *feedback* em tempo real, demonstrando alta eficiência em simulações multiescala massivas. Essas abordagens são particularmente eficazes em cenários que exigem coordenação de milhares de *jobs* simultâneos e gerenciamento de grandes volumes de dados, como em simulações científicas complexas.

Diferentemente, a Cloud AutoDroid foca em um nicho específico de aplicações, como a MalSynGen, que é uma aplicação parametrizável para execução de modelos específicos de domínio. Enquanto *frameworks* como MLflow e MetaFlow são voltados para *workflows* genéricos de ML, e soluções como MuMMI são projetadas para simulações multiescala, a Cloud AutoDroid é otimizada para aplicações "standalone" e auto-contidas. Essas aplicações são caracterizadas por receberem um conjunto de dados como entrada e produzirem outro conjunto de dados como saída, sem a necessidade de integração complexa com outros sistemas ou *workflows*. Essa abordagem permite que a Cloud AutoDroid atenda a demandas específicas de geração de dados sintéticos e experimentação em larga escala, como no caso da MalSynGen, com foco em eficiência e escalabilidade para domínios especializados.

3. Cloud AutoDroid: Arquitetura e Implementação

A Cloud AutoDroid oferece um alto nível de abstração para executar ferramentas de IA de ponta com conjuntos de dados expressivos, como a MalSynGen, disponibilizando-as por meio de um serviço distribuído e escalável. A plataforma é resultado de um ciclo completo de engenharia de software [Laviola et al. 2024].

A ferramenta combina três componentes principais: a API, os *workers* e um Sistema de Telemetria, conforme ilustrado na Figura 1. A API é responsável exclusivamente pelo gerenciamento e atendimento das requisições, enquanto os *workers* executam as cargas de trabalho recebidas. O Sistema de Telemetria monitora e coleta métricas em tempo real, permitindo uma análise detalhada do processamento distribuído, tanto durante a execução quanto em análises *post-mortem*, garantindo maior visibilidade e controle sobre o desempenho do sistema.

O isolamento e a segurança são requisitos fundamentais na arquitetura da Cloud AutoDroid, pois cada nó opera em um ambiente independente, reduzindo riscos de acesso não autorizado e protegendo a infraestrutura da API. A solução também possibilita a redução de custos operacionais, permitindo a utilização de máquinas já disponíveis localmente para alocação de nós, evitando a dependência exclusiva de serviços de nuvem. Por fim, a arquitetura garante alta disponibilidade, assegurando que, caso um *worker* não consiga adquirir uma tarefa de processamento, ela seja redirecionada automaticamente para outro nó disponível. Essa combinação de características torna a Cloud AutoDroid uma solução robusta e eficiente para a execução de experimentos de IA em larga escala.

Inspirada na abordagem de computação distribuída do *Self-Hosted Runners do GitHub Actions*[2], a Cloud AutoDroid traz benefícios operacionais significativos, como a possibilidade de instanciar novos nós de processamento sem expor a infraestrutura interna da API. Para isso, basta um computador com acesso à internet e os

[2]`https://docs.github.com/en/actions/hosting-your-own-runners`

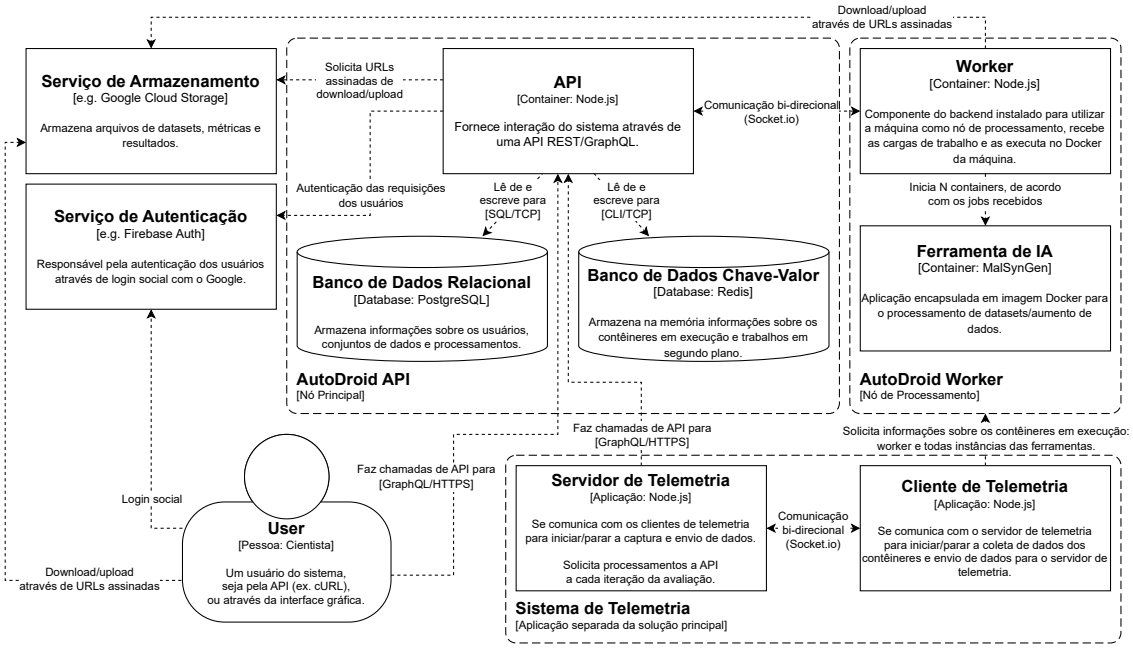


Figura 1. Arquitetura da Cloud AutoDroid.

requisitos mínimos de processamento para ferramentas como a MalSynGen. Além disso, a abordagem oferece escalabilidade flexível, permitindo a criação de novos nós de processamento sem a necessidade de configurações complexas.

A API é responsável por receber todas as requisições, tanto externas (dos usuários) quanto internas (dos *workers*). Ela gerencia o fluxo de dados desde as camadas de apresentação — que incluem *controllers* REST e *resolvers* GraphQL —, passando pelas camadas de serviços, onde as regras de negócio são implementadas, até a camada de persistência, responsável pela comunicação com os bancos de dados. A API incorpora elementos do *Domain-Driven Design* (DDD) e segue os princípios SOLID (Responsabilidade Única, Aberto/Fechado, Substituição de Liskov, Segregação de Interfaces e Inversão de Dependência). Desenvolvida em Typescript[3] e executada com o *runtime* Node.js[4], suas regras de negócio são validadas por meio de testes automatizados — incluindo testes de integração, unitários e ponta a ponta (E2E) — utilizando o *framework* Vitest[5] e a biblioteca Testcontainers[6].

A *worker*, por sua vez, atua como nó de processamento, permitindo a descentralização de tarefas. Ela se comunica com a API por meio de uma conexão bidirecional fornecida pelo Socket.io e inicia a ferramenta solicitada no ambiente Docker da máquina hospedeira. A *worker* foi implementada na mesma linguagem e *runtime* da API, garantindo consistência e facilidade de manutenção.

A Cloud AutoDroid inclui instrumentos para coletar informações sobre os *workers*. Embora *frameworks* de observabilidade, como o *OpenTelemetry*, tenham sido considerados, optamos pelo desenvolvimento de um sistema de tele-

[3]`https://www.typescriptlang.org/`
[4]`https://nodejs.org/`
[5]`https://vitest.dev/`
[6]`https://testcontainers.com/`

metria no modelo cliente[7]/servidor[8] próprio. Essa decisão permite maior controle sobre o sistema e minimiza interferências, como o efeito conhecido de *agent overhead* [Kuba 2023], além de manter o sistema de telemetria desacoplado do núcleo da Cloud AutoDroid.

O Sistema de Telemetria utiliza a mesma *stack* tecnológica da Cloud AutoDroid — Typescript e Node.js. No entanto, para reduzir o tamanho em disco e a degradação no desempenho, foram adotadas apenas as dependências essenciais, seguidas pela transformação do código para JavaScript puro. A aplicação cliente utiliza a biblioteca *systeminformation*[9] para capturar as informações do *host*, e a biblioteca *dockerode*[10] para monitorar os contêineres em execução e suas métricas de desempenho. A aplicação servidor agrega as funcionalidades necessárias para receber e tratar os dados, além de comandos para o cálculo estatístico e a geração de gráficos. Toda a comunicação entre clientes e servidor é realizada utilizando a biblioteca *Socket.io*[11], garantindo um canal de comunicação bidirecional em tempo real.

4. Avaliação

4.1. Metodologia

Na avaliação foram utilizadas 11 máquinas, sendo uma 1 para executar a API e outras 10 para executar *workers*. Essas máquinas possuem configurações descritas na Tabela 1. Máquinas com mesma responsabilidade e configuração de hardware foram organizadas em grupos. Cada máquina executou apenas um *worker* ou API, além do cliente ou servidor de telemetria.

Tabela 1. Relação dos grupos de nós e suas especificações.

Grupo	Nós	CPU	Mem.	OS	Kernel	Largura de rede
API	1	Intel® Core™ i7-9700	16GB	Debian 12	6.1.0-27-amd64	1GB
G1	4	Intel® Xeon™ E312xx (2 vCPU)	6GB	Ubuntu 22.04.3	5.15.0-89-generic	1GB (compartilhado)
G2	4	Intel® Core™ i7-9700	16GB	Debian 12	6.1.0-27-amd64	1GB
G3	1	Intel® Core™ i7-12650H	32GB	Debian 12	6.1.0-31-amd64	1GB
G4	1	AMD® Ryzen™ 7 5800X	84GB	Ubuntu 24.04.2	6.8.0-49-generic	1GB

A Figura 2 ilustra a topologia da rede utilizada no experimento. As máquinas dos grupos 2 a 4 estavam localizadas no mesmo domínio que a API, enquanto as máquinas do grupo 1 estavam em um domínio externo (todas no mesmo domínio). As latências médias, obtidas após 100 execuções de medições com ICMP entre os nós e a API, são apresentadas na Tabela 2. Podemos observar que as latências são similares tanto para os 6 *workers* localizados no mesmo domínio da API (A) quanto para os 4 em domínios externos (B).

O experimento foi organizado em iterações. Em cada iteração ($i \in I = \{1, \ldots, 3\}$), foram solicitados $T = i \times W$ tarefas de processamento, onde $W = 10$ é

[7] `https://github.com/MalwareDataLab/autodroid-watcher-client`
[8] `https://github.com/MalwareDataLab/autodroid-watcher-server`
[9] `https://www.npmjs.com/package/systeminformation`
[10] `https://www.npmjs.com/package/dockerode`
[11] `https://socket.io`

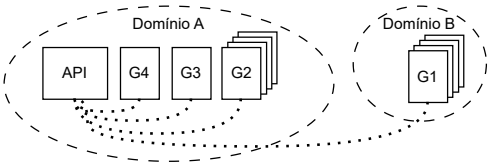


Figura 2. Topologia da rede.

Tabela 2. Latência Worker-API.

Domínio	Grupo	Máq.	Latência
B	G1	M1	42.232 ms
B	G1	M2	42.192 ms
B	G1	M3	42.212 ms
B	G1	M4	42.172 ms
A	G2	M1	0.517 ms
A	G2	M2	0.523 ms
A	G2	M3	0.507 ms
A	G2	M4	0.513 ms
A	G3	M1	0.688 ms
A	G4	M1	0.855 ms

a quantidade de *workers*. Usamos uma política de balanceamento uniforme, onde cada *worker* recebe exatamente a mesma quantidade de tarefas (T/W). Uma nova iteração pode ser iniciada somente após a conclusão completa da iteração anterior. Em todas iterações usamos uma tarefa igual, isto é, com mesmo *dataset*, ferramenta de IA (MalSynGen) e conjunto de hiperparâmetros. Carregamos a imagem da ferramenta de IA no *worker* previamente ao início do experimento.

O servidor do Sistema de Telemetria é responsável por solicitar à API e enviar o sinal de início de cada etapa aos clientes, que passam a coletar informações a cada segundo. Esse processo continua até que o servidor confirme a conclusão de todas as tarefas (T) da etapa, interrompendo a coleta e reiniciando a contagem para a próxima iteração. Ao final de todas as etapas, comandos específicos são executados para calcular estatísticas e gerar gráficos. O servidor de telemetria, operando em uma máquina distinta, utiliza o serviço de túnel HTTP *Cloudflare Tunnel*[12] para receber os dados dos clientes.

O comando de início do experimento é enviado paralelamente para todos os clientes, que iniciam a coleta e o envio dos dados para o servidor de telemetria. Este, por sua vez, registra os dados em arquivos CSV até que todos os processamentos sejam concluídos, encerrando a iteração.

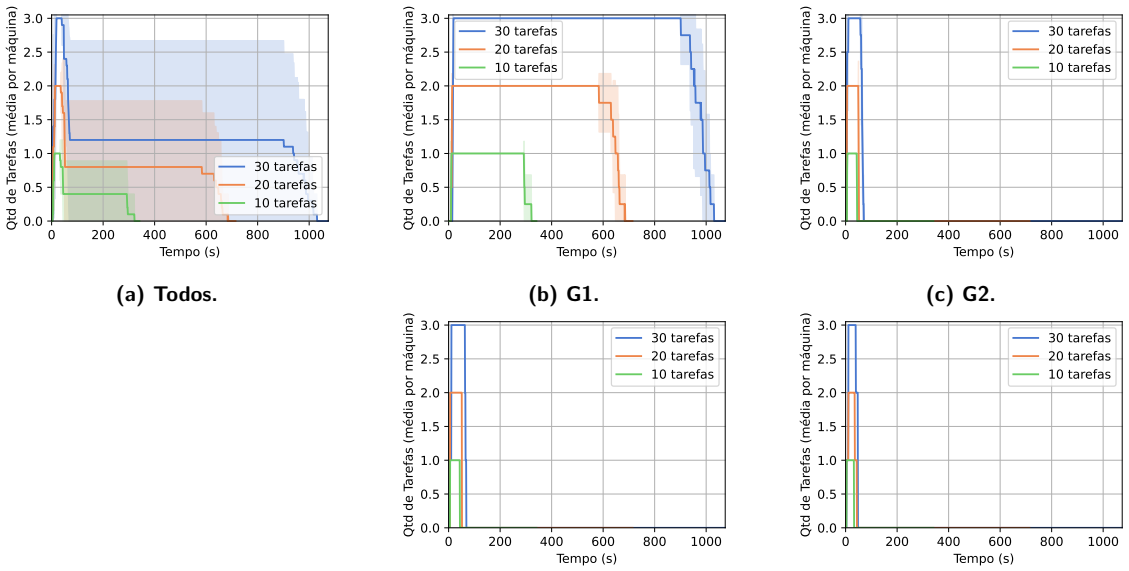
4.2. Resultados

A Figura 3 apresenta, para cada iteração, a média (linha) e o desvio padrão (área sombreada) da quantidade de tarefas em execução nos *workers* ao longo do tempo, considerando 5 recortes distintos (todos os grupos e cada grupo isolado). Em geral, podemos observar que o número de atividades em execução atinge o máximo esperado para cada iteração, conforme projetado pelo balanceamento uniforme.

A Figura 3a mostra que o tempo máximo de uma iteração ($i = 3$) do experimento é de aproximadamente 1.000 segundos e que o desvio padrão da quantidade de tarefas executadas em paralelo é elevado. Para entender esse fenômeno, os dados foram isolados em cada um dos grupos nas Figuras 3b, 3c, 3d e 3e. Nelas podemos observar que o desvio padrão é relativamente baixo, e o G1 leva um tempo significativamente maior que os outros grupos para executar as mesmas tarefas.

A Figura 4 apresenta o uso de *CPU* e memória ao longo dos experimentos da iteração $i = 3$ (30 tarefas). Podemos notar que a *CPU* é um gargalo para o G1, mas não representa um gargalo para os outros grupos. Já a memória não se configura como um gargalo para nenhum dos grupos.

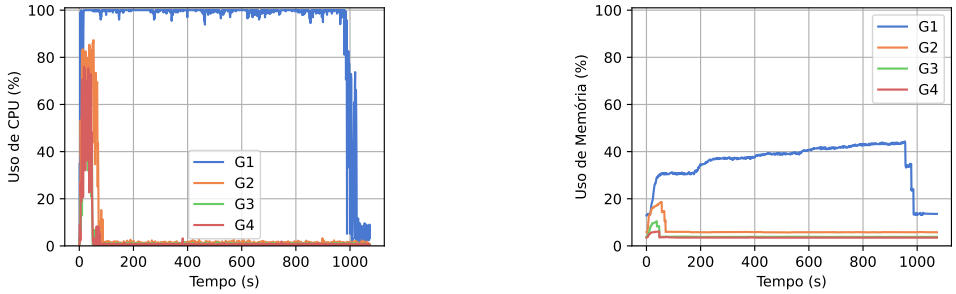
[12] `https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/`



(a) Todos. (b) G1. (c) G2.

(d) G3. (e) G4.

Figura 3. Tarefas em execução ao longo do tempo.

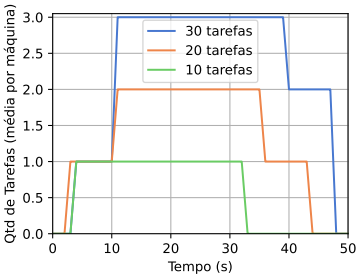


(a) CPU. (b) Memória.

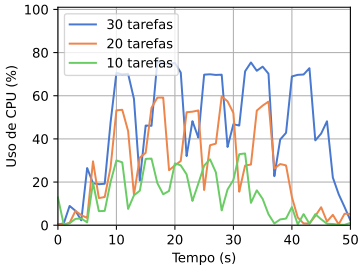
Figura 4. Uso de Recursos (uma máquina de cada grupo), iteração $i = 3$ (30 tarefas).

A Figura 5 apresenta evidências detalhadas sobre o G4. É importante ressaltar que consideramos um trabalho em execução somente a partir do momento em que o contêiner é iniciado. Podemos observar na Figura 5a que o início efetivo ocorre a partir de 5 segundos. No entanto, entre a solicitação e o início do contêiner, diversos processos ocorrem, como a validação da solicitação, a seleção e o encaminhamento da tarefa, além dos procedimentos de preparação e verificação realizados pelo *worker*. Com base na Figura 5b, que revela um aumento no uso da *CPU* por volta de 5 segundos, estimamos que o *worker* começa a receber a carga entre 2 e 3 segundos após a solicitação, assumindo uma fila vazia e *workers* ociosos. Consideramos que esses valores são aceitáveis para as cargas de trabalho previstas, que podem levar horas para serem concluídas.

A Figura 6 mostra a média e o desvio padrão do uso de memória em função do número de tarefas em execução. Como esperado, observamos que os valores podem ser aproximados por um polinômio de primeira ordem e que o fator de inclinação das curvas é similar para os diferentes sistemas. A constante inicial é diferente, possivelmente devido a alguma política de sistema com recursos mais restritos.



(a) Tarefas.



(b) CPU.

Figura 5. Tarefas e uso de CPU no G4.

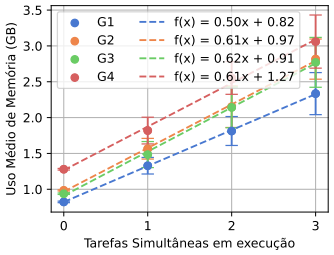


Figura 6. Memória.

5. Considerações Finais

A Cloud AutoDroid representa um avanço significativo na execução de experimentos de *IA*, oferecendo uma solução escalável que simplifica a curva de aprendizado e reduz os requisitos técnicos de utilização. Suas principais contribuições incluem: (a) escalabilidade através de uma arquitetura distribuída e assíncrona, permitindo a expansão de nós em diferentes ambientes; (b) gerenciamento integrado com controle de *datasets*, ferramentas, validação e monitoramento; e (c) acessibilidade a ambiente útil tanto para iniciantes e estudantes quanto para especialistas.

Como trabalhos futuros podemos destacar: otimização da distribuição de carga para reduzir o tempo de fila; avaliação com diferentes conjuntos de dados, ferramentas e configurações; desenvolvimento de algoritmos de balanceamento de carga considerando *GPUs* e disponibilidade de recursos; avaliação da escalabilidade da API em cenários complexos; e investigação de custos operacionais e integração com serviços em nuvem para autoescalabilidade plena.

Demonstração. Para executar a demonstração é necessário um computador com sistema operacional Linux (Ubuntu, Debian ou similar). A virtualização deve estar habilitada na BIOS, com pelo menos 8GB de RAM e 10GB de espaço livre em disco para armazenamento de arquivos, resultados, banco de dados e imagens Docker. Além disso, é essencial ter o Git e o Docker instalados para gerenciamento de código e execução de contêineres.

Artefatos. Todas as informações necessárias para reproduzir esta avaliação, incluindo a ferramenta e documentação, estão disponíveis em um repositório público[13], juntamente com todos os artefatos.

Agradecimentos. A pesquisa contou com apoio da RNP (Programa Hackers do Bem - GT Malware DataLab), da CAPES (Código de Financiamento 001), da Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul (FAPERGS), por meio dos editais 02/2022, 08/2023, 09/2023 e dos termos de outorga 24/2551-0001368-7 e 24/2551-0000726-1, e da FAPESP (processos 2020/05183-0 e 2023/00816-2).

Referências

Bhatia, H., Di Natale, F., Moon, J. Y., Zhang, X., Chavez, J. R., Aydin, F., Stanley, C., Oppelstrup, T., Neale, C., Schumacher, S. K., Ahn, D. H., Herbein, S., Car-

[13] `https://github.com/MalwareDataLab/autodroid-sbrc25`

penter, T. S., Gnanakaran, S., Bremer, P.-T., Glosli, J. N., Lightstone, F. C., and Ingólfsson, H. I. (2021). Generalizable coordination of large multiscale workflows: challenges and learnings at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA. Association for Computing Machinery.

Bragança, H., Rocha, V., Barcellos, L., Souto, E., Kreutz, D., and Feitosa, E. (2023). Capturing the behavior of android malware with mh-100k: A novel and multidimensional dataset. In *Anais do XXIII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 510–515, Porto Alegre, RS, Brasil. SBC.

Chen, K., Lee, Y., and Soh, H. (2021). Multi-modal mutual information (mummi) training for robust self-supervised deep reinforcement learning. In *2021 IEEE international conference on robotics and automation (ICRA)*, pages 4274–4280. IEEE.

Krawczuk, P., Papadimitriou, G., Tanaka, R., Anh Do, T. M., Subramanya, S., Nagarkar, S., Jain, A., Lam, K., Mandal, A., Pottier, L., and Deelman, E. (2021). A performance characterization of scientific machine learning workflows. In *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 58–65.

Kuba, M. (2023). Otel component performance benchmarks. Disponível em: `https://opentelemetry.io/blog/2023/perf-testing/`.

Laviola, L., Paim, K., Kreutz, D., and Mansilha, R. (2023). Autodroid: disponibilizando a ferramenta droidaugmentor como serviço. In *Anais da XX Escola Regional de Redes de Computadores*, pages 145–150, Porto Alegre, RS, Brasil. SBC.

Laviola, L. F., Nogueira, A. G. D., Kreutz, D., and Mansilha, R. B. (2024). Cloud autodroid: uma arquitetura de backend para executar serviços de ia generativa na nuvem. In *Anais da VIII Escola Regional de Engenharia de Software*, pages 258–267, Porto Alegre, RS, Brasil. SBC.

Nogueira, A., Paim, K., Bragança, H., Mansilha, R., and Kreutz, D. (2024a). Malsyngen: redes neurais artificiais na geração de dados tabulares sintéticos para detecção de malware. In *Anais Estendidos do XXIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 129–136, Porto Alegre, RS, Brasil. SBC.

Nogueira, A. G. D., de Oliveira, L. F. A., da Silva, A. L. G., Kreutz, D., and Mansilha, R. B. (2024b). Otimização de hiperparâmetros da droidaugmentor para geração de dados sintéticos de malware android. In *Anais da XXI Escola Regional de Redes de Computadores / VIII Workshop de Segurança (ERRC/WRSeg)*. Disponível em: `https://drive.google.com/file/d/1UpEzupoBrjgiIaJp6EzFQJVyhpJi6FGP/view`.

Polyxronou, I. (2024). Designing a scalable and reproducible machine learning workflow. Master's thesis, University of West Attica.

Tagliabue, J., Bowne-Anderson, H., Tuulos, V., Goyal, S., Cledat, R., and Berg, D. (2023). Reasonable scale machine learning with open-source metaflow.

Wozniak, J. M., Davis, P. E., Shu, T., Ozik, J., Collier, N., Parashar, M., Foster, I., Brettin, T., and Stevens, R. (2018). Scaling deep learning for cancer with advanced workflow storage integration. In *Proceedings of Machine Learning in High Performance Computing Environments*, Argonne, IL, USA. Argonne National Laboratory. Available at: `https://mcs.anl.gov`.

Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., et al. (2018). Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45.