ChatGPT

# Guia Técnico: Instalação e Uso do NVIDIA Isaac GR00T N1.5-3B no Pipeline Isaac Sim/Isaac Lab

Este guia descreve passo a passo como instalar, configurar e utilizar o modelo **GR00T N1.5-3B** em seu ambiente, integrando-o ao pipeline existente (validado até o *Step 4*) com Isaac Sim 5.0 e Isaac Lab. O objetivo é garantir reprodutibilidade e compatibilidade com seu setup atual (Ubuntu 24.04, GPU RTX 4070, robô Unitree G1 convertido para USD com 23 DOFs, teleoperação SE(2) via Isaac Lab no ambiente `Isaac-Velocity-Flat-G1-Play-v0`). Também incluímos dicas para verificar o funcionamento correto e orientações para futura integração via interface de controle planar (SE(2)). Siga os passos abaixo em sequência:

## 1. Preparação do Ambiente e Dependências

- **Ative o ambiente Conda já existente** (que contém Isaac Sim/Lab). Por exemplo: `conda activate <nome_do_env>`. Certifique-se de estar usando Python **3.10**, pois o GR00T foi testado com essa versão [1] [2].
- **Instale bibliotecas de sistema ausentes:** O GR00T requer codecs e libs gráficas. Atualize os pacotes e instale **FFmpeg** e libs de GUI:

  ```
  sudo apt update && sudo apt install -y ffmpeg libsm6 libxext6 libgl1-mesa-glx libglib2.0-0
  ```

  Esses pacotes garantem suporte a processamento de vídeo/imagens (por exemplo, `ffmpeg`, `libsm6`, `libxext6`) conforme as dependências listadas [3].
- **Verifique o CUDA e drivers NVIDIA:** O GR00T N1.5 foi desenvolvido para GPUs NVIDIA (Ampere/Lovelace ou superior) em Linux [4]. Recomenda-se ter **CUDA 12.4** instalado para evitar problemas ao compilar módulos como FlashAttention [5]. Confirme com `nvidia-smi` se a GPU é reconhecida e com `nvcc --version` (se disponível) a versão do CUDA. Se necessário, atualize os drivers e instale o CUDA 12.4 seguindo as instruções da NVIDIA [1]. *(Observação: Isaac Sim 5.0 geralmente já exige drivers recentes, então este passo é apenas para garantir compatibilidade total.)*

## 2. Instalação do Isaac-GR00T no Ambiente Conda

1. **Obtenha o código-fonte do GR00T:** Navegue até um diretório de sua preferência (por exemplo, `~/projects` ou o local onde organiza pacotes do Isaac Lab) e clone o repositório GitHub oficial:

   ```
   git clone https://github.com/NVIDIA/Isaac-GR00T.git
   cd Isaac-GR00T
   ```

Isso baixa o projeto e posiciona você no diretório do mesmo [6].

2. **Atualize ferramentas Python (opcional):** Atualize `setuptools` (e opcionalmente o pip) para evitar problemas de instalação:

```
pip install --upgrade setuptools
```

(Caso seu pip seja antigo, `pip install --upgrade pip`.)

3. **Instale as dependências Python do GR00T:** Execute o pip install editável com os requisitos *base*:

```
pip install -e .[base]
```

Isso irá instalar o pacote `gr00t` e bibliotecas necessárias (PyTorch, Transformers, HuggingFace Hub etc.) no seu ambiente [7]. Aguarde a conclusão e verifique se não houve erros.

4. **Instale FlashAttention otimizado:** O GR00T tira proveito de um kernel otimizado de atenção. Instale a versão recomendada do FlashAttention:

```
pip install --no-build-isolation flash-attn==2.7.1.post4
```

Essa instalação pode baixar um wheel pré-compilado compatível com CUDA 12.4/PyTorch 2.5.1. Caso não exista wheel para seu setup, o pip irá compilar do fonte, o que **pode ser demorado** [8]. *Dicas:* Se a compilação estiver muito lenta ou falhar, você pode tentar baixar o wheel pré-compilado correspondente do repositório oficial do FlashAttention e instalar via `pip install <arquivo.whl>` [8]. Certifique-se também de que sua versão do PyTorch é compatível (a combinação PyTorch 2.5.1 + CUDA 12.4 é conhecida por funcionar [9] – se necessário, instale especificamente `torch==2.5.1 torchvision==0.20.1` para alinhar versões).

5. **Correções de instalação (se necessárias):** Em versões antigas do repositório, houve um problema com a dependência **PyAV** nomeada incorretamente. Se o comando acima falhar com erro relacionado a `pyav`, edite o arquivo `pyproject.toml` do GR00T e substitua `"pyav"` por `"av"`, então reinstale [10]. Alternativamente, instale manualmente o pacote AV com `pip install av` antes de rodar o `pip install -e .[base]`.

6. **Valide a instalação:** Após a instalação, verifique que o pacote foi instalado corretamente. Você pode abrir um interpretador Python (`python`) e tentar `import gr00t` para ver se carrega sem erros. Também confira se o comando `pip list` mostra `gr00t` e dependências (transformers, etc.) instaladas.

## 3. Download e Cache do Modelo GR00T N1.5-3B

Antes de executar o modelo, vamos baixar os pesos pré-treinados do GR00T N1.5-3B (aprox. 2,72 bilhões de parâmetros) para uso local. O modelo está hospedado no HuggingFace Hub [11] sob o nome `"nvidia/GR00T-N1.5-3B"`.

- **Baixe os pesos via HuggingFace:** Certifique-se de estar conectado à internet e com espaço em disco (~6 GB disponíveis). No próprio ambiente, use o utilitário do HuggingFace para baixar todos os arquivos do modelo de uma vez:

```
python -c "from huggingface_hub import snapshot_download;
snapshot_download('nvidia/GR00T-N1.5-3B')"
```

Esse comando fará o download dos *shards* do modelo (arquivos `.safetensors`) e outros arquivos necessários, armazenando-os no cache local do HuggingFace (tipicamente em `~/.cache/huggingface/`). Aguarde até o término (pode demorar alguns minutos, dependendo da conexão).

- **(Opcional) Autenticação no HuggingFace:** O modelo GR00T N1.5 é de uso aberto (licença NSCL v1), não requer token de autenticação para download público. Porém, se você tiver problemas de acesso ou quiser usar um token (por ex., para retomar downloads), faça login com `huggingface-cli login` e tente novamente.
- **Verifique o cache completo:** Após baixar, verifique se os arquivos principais (por exemplo, `model-00001-of-00004.safetensors` até `...00004-of-00004.safetensors`, e o `config.json`, etc.) estão presentes em `~/.cache/huggingface/hub` para o modelo. Isso garante que o servidor não enfrentará atrasos no primeiro uso devido a download.
- **Modelo alternativo mais leve:** Caso sua GPU tenha menos memória ou você queira testar performance, há uma versão anterior **GR00T N1-2B** (~2 bilhões de parâmetros) disponível [12]. Para usá-la, baixe `"nvidia/GR00T-N1-2B"` e use esse caminho de modelo nas etapas seguintes. Lembre-se, porém, que o N1.5-3B tem melhorias significativas em compreensão de linguagem e desempenho [13] [14], portanto use o modelo menor apenas se necessário.

## 4. Inicialização do Servidor de Inferência GR00T

Com o ambiente configurado e o modelo baixado, podemos iniciar o servidor de inferência do GR00T. O servidor carregará o modelo na GPU (se disponível) e aguardará requisições de ação. Siga os passos:

1. **Confirme ambiente e diretório:** Ative o ambiente conda correto (`conda activate ...`) e certifique-se de estar no diretório raiz do projeto `Isaac-GR00T` (onde fica a pasta `scripts/`).
2. **Execute o servidor GR00T:** Rode o script de serviço de inferência em modo servidor:

   ```
   python scripts/inference_service.py --server
   ```

   Esse comando inicia o servidor padrão (baseado em ZMQ) na porta **5555**, usando por padrão o modelo `nvidia/GR00T-N1.5-3B`, *embodiment tag* `"gr1"` e configuração de dados `"fourier_gr1_arms_waist"` [15] [16]. Você deverá ver mensagens no console indicando o carregamento do modelo e inicialização do servidor, por exemplo: *"Policy loaded from nvidia/GR00T-N1.5-3B"* e *"Server started on localhost:5555"*. *(Dica: a primeira inicialização pode demorar um pouco enquanto os pesos do modelo são carregados na memória GPU.)*
3. **Uso de modelo alternativo:** Para usar o modelo leve N1-2B ou um checkpoint custom, adicione `--model-path <nome_ou_caminho_do_modelo>` ao comando. Ex: `--model-path nvidia/GR00T-N1-2B`.
4. **Modo HTTP (opcional):** Por padrão, o servidor usa sockets ZMQ internos. Se preferir expor uma API REST (por exemplo, para integrar via HTTP), você pode iniciar com `--http-server --port 8000` (nesse caso instale dependências extras: `pip install uvicorn fastapi json-numpy` conforme indicado no cabeçalho do script [17]). Para esta integração inicial, o modo padrão (ZMQ) é suficiente.

5. **Acompanhe o uso da GPU:** Enquanto o servidor carrega, abra outra janela de terminal e execute `nvidia-smi`. Você deverá ver um processo Python consumindo memória GPU (alguns GB, correspondentes aos ~2.72B parâmetros do modelo) [18]. Isso confirma que o modelo está na GPU. O GR00T é otimizado para acelerar em GPUs NVIDIA e usa cálculos em baixa precisão (BF16) para eficiência [4]. Se por algum motivo a GPU não for utilizada (por exemplo, ausência de CUDA disponível), o modelo cai para CPU, mas isso resultará em inferência muito mais lenta.
6. **Verifique a saída do servidor:** No terminal do servidor, verifique se não há erros. Você deve ver logs similares a:
7. Mensagem de modelo carregado e configuração usada (embodiment tag e data config).
8. Porta e host onde o servidor está escutando.
   Se houve erro (por ex., **porta ocupada** ou **problema ao baixar/carregar modelo**), consulte a seção de Depuração no final deste guia para soluções. Caso tudo esteja OK, mantenha este terminal do servidor aberto (o servidor rodando) para o próximo passo.

## 5. Teste do Servidor com Cliente Local

Com o servidor de inferência em execução, vamos testá-lo usando o modo cliente fornecido, garantindo que ele responde corretamente. O cliente de teste envia uma observação simulada (aleatória) e imprime a ação recebida do modelo.

1. **Abra um novo terminal e ative o ambiente** (para isolar do terminal do servidor, mas mantendo o servidor rodando). Navegue novamente até o diretório `Isaac-GR00T`.
2. **Execute o cliente de teste:** Rode o comando:

   ```
   python scripts/inference_service.py --client
   ```

   Este comando faz o script entrar em modo cliente, conectando-se ao servidor (padrão `localhost:5555`) e enviando um pacote de observação de exemplo [19] [20]. Por padrão, o cliente gera dados randômicos que simulam entradas do robô: uma imagem *dummy* (`video.ego_view` de 256x256 pixels), vetores de estado aleatórios para braços/mãos (`state.left_arm`, `state.right_arm`, etc.) e um comando de texto fictício (`"do your thing!"`) [20].
3. **Observe a resposta:** No terminal cliente, você deverá ver uma mensagem indicando o tempo total da inferência e a ação retornada. Algo como: *"Total time taken to get action from server: X.XX seconds"* e em seguida um objeto/array representando a ação contínua calculada [21]. A estrutura exata da ação depende do *embodiment* usado (no caso padrão `gr1`, será um vetor de valores correspondentes às articulações dos braços, mãos e cintura do robô *GR-1* humanoide).
4. **Valide o funcionamento:** A presença de uma resposta numérica do servidor indica que o pipeline de inferência está funcionando. Verifique no terminal do servidor que ele registrou a requisição e possivelmente imprimiu alguma informação (como *"Available modality config..."* e a lista de chaves de observação recebidas). Além disso, verifique novamente o `nvidia-smi` durante a inferência – deve ter havido uso de GPU (picos de uso de GPU e VRAM estável).
5. **Teste adicional (opcional):** Você pode rodar múltiplas vezes o cliente para ver a resposta variando (devido aos inputs aleatórios). Se preferir, pode editar o script para testar inputs personalizados, mas isso não é necessário nesta validação inicial. O importante é confirmar que o servidor responde sem erros.

6. **Encerrar teste:** Após verificar o funcionamento, você pode fechar o terminal cliente. Deixe o servidor rodando se for realizar integrações a seguir. Para parar o servidor, use `Ctrl+C` no terminal do servidor (ele encerrará o processo do modelo).

## 6. Verificações de Funcionamento (Checklist de Validação)

Antes de prosseguir para integração, confira estes pontos para garantir que tudo está funcionando conforme esperado:

- **Resposta do Modelo:** O servidor retornou uma ação válida ao cliente de teste (vetor numérico). Isso confirma que o GR00T está executando inferências.
- **Uso da GPU:** Confirme que o modelo carregou na GPU (verificado via `nvidia-smi`). Em caso de uso incorreto (ex.: modelo rodando em CPU ou não alocando GPU), revise a instalação do PyTorch/CUDA. O modelo GR00T N1.5 foi projetado para tirar vantagem de GPUs NVIDIA modernas, obtendo inferências em torno de ~50 ms em uma RTX 4090 [22] (o que deve ser da mesma ordem na RTX 4070, talvez ligeiramente mais lento). Em CPU, esse tempo pode subir para segundos.
- **Consumo de Memória:** Verifique se o uso de VRAM (~5-6 GB esperado) cabe na sua GPU de 12 GB. Se a GPU estiver no limite ou se ocorrer erro *out-of-memory*, considere reduzir exigências: diminuir `--denoising-steps` (p.ex., de 4 para 2) para reduzir uso temporário [22] [23], ou usar o modelo menor N1-2B.
- **Integridade do Download:** Se o modelo não carrega ou dá erro de arquivo corrompido, pode ser download incompleto. Remova o cache do modelo (`~/.cache/huggingface/hub/.../GR00T-N1.5-3B`) e refaça o download, garantindo conclusão sem interrupções.
- **Porta de Servidor:** O servidor default usa porta 5555. Se o comando `--server` não mostrou *"Server started on ..."* e simplesmente retornou ao prompt, possivelmente a porta estava ocupada (ou outra instância em execução). Nesse caso, tente lançar em outra porta: `python scripts/inference_service.py --server --port 5556` (e ajuste o cliente com `--port 5556` também) [24].
- **Logs de Erro/Incompatibilidades:** Fique atento a avisos no terminal do servidor ao carregar o modelo. Qualquer *traceback* deve ser resolvido antes de prosseguir. Por exemplo, erro de *flash-attn* ausente ou versão CUDA incompatível: reinstale conforme seção de instalação (ou tente fallback usando `pip install xformers` como alternativa de backend de atenção, se necessário). Um aviso comum pode ser sobre `transformer_engine` – no Docker oficial eles removem esse pacote para evitar conflitos [25], mas se não houve erro na nossa instalação, não se preocupe.
- **Desempenho:** Mesmo sem otimizações adicionais (TensorRT, etc.), o modelo deve rodar em tempo razoável. Caso pretenda uso em tempo real, verifique se a latência medida está dentro do aceitável para sua aplicação. O GR00T N1.5 com 4 passos de difusão alcança inferência ~50ms em GPUs topo de linha [22]; na RTX 4070 espere possivelmente ~70-100ms. Tempos muito acima disso sugerem fallback no CPU ou algum gargalo – confira se flash-attn está funcionando (senão, a atenção padrão pode ser mais lenta).

## 7. Diretrizes para Integração com Isaac Lab (Controle SE(2))

Com o servidor GR00T validado, você pode começar a integrá-lo ao seu sistema de simulação para substituir ou complementar a teleoperação por inteligência do modelo. A abordagem sugerida é uma arquitetura **desacoplada**, mantendo o **endpoint de inferência** (servidor GR00T) separado do **endpoint de controle** (Isaac Lab), comunicando-se via cliente-servidor [26]. Isso traz modularidade e permite que o

pesado cálculo de IA ocorra independentemente do loop de simulação. Seguem diretrizes para realizar essa integração:

- **Escolha do Modo de Comunicação:** Decida como o Isaac Lab vai solicitar ações ao GR00T:
- *Via ZMQ (Python)*: Importando as classes do GR00T no código do Isaac Lab e usando o cliente nativo. Por exemplo, no script de controle do Isaac Lab você pode `from gr00t.eval.robot import RobotInferenceClient` e conectá-lo ao servidor (`client = RobotInferenceClient(host="localhost", port=5555)`). Assim, pode chamar `action = client.get_action(obs_dict)` para obter ações. Essa abordagem requer que o ambiente Isaac Lab rode no **mesmo ambiente Python** com o pacote `gr00t` instalado (no nosso caso, sim).
- *Via HTTP (REST)*: Executando o servidor GR00T em modo HTTP (`--http-server`) e, no lado Isaac Lab, usando requisições web. Por exemplo, usar a biblioteca `requests` para fazer POST no endpoint `/act` do servidor FastAPI do GR00T [27] [28]. Esse método é útil se quiser manter ambientes separados (e.g., Isaac Sim em outra máquina ou container), mas introduz latência de serialização. No seu caso local, o método ZMQ/Python direto é perfeitamente viável e mais eficiente.
- **Coleta de Observações do Simulador:** No loop de simulação (por exemplo, dentro do *task* ou *env step* do Isaac Lab), colete os dados de estado necessários e formate-os no **dicionário de observação esperado pelo GR00T**. Isso inclui:
- **Imagem(s) de câmera**: se seu robô/simulação fornece câmera RGB, obtenha o frame atual. Converta para dimensão `[1, H, W, 3]` (com H=W=224 ou 256, dependendo do treinado – GR00T N1.5 usa 224x224 em treinamento, mas no exemplo do cliente usou 256x256 [20]; manter 256x256 é seguro). Nomeie a entrada como `"video.ego_view"` ou outra chave conforme o *modality config* utilizado. Para consistência, você pode usar `"video.ego_view"` se estiver usando o config padrão do GR1.
- **Estados proprioceptivos (juntas)**: extraia os valores das articulações relevantes. Por exemplo, se o modelo estiver usando o *embodiment* `gr1` (humanoide com braços e mãos), ele espera estados como `"state.left_arm"` (tamanho 7), `"state.right_arm"` (7), `"state.left_hand"` (6), `"state.right_hand"` (6) e `"state.waist"` (3) [20]. No seu caso (Quadrúpede Unitree com braços Inspire), você precisará decidir como mapear: talvez usar os braços do robô mapeados para `left_arm/right_arm`, os atuadores das mãos para `left_hand/right_hand`, e usar `waist` para algum grau de liberdade do tronco se houver. Se o base (pernas do quadrúpede) não for controlado pelo GR00T neste momento, pode mantê-las fixas ou ignoradas (o config `fourier_gr1_arms_waist` de fato ignora pernas do humanoide, focando em braços/cintura). Ajuste o obs_dict de acordo, preenchendo as chaves esperadas pelo modelo com seus dados (ou zeros se irrelevante).
- **Instrução de linguagem (goal):** Formule o comando desejado em linguagem natural e passe na chave `"annotation.human.action.task_description"`. Este é o campo de *prompt* que o modelo utiliza para condicionar a ação [29]. Por exemplo: `"Pegue o objeto vermelho da mesa"` ou `"Ande para frente 1 metro"`. Como seu foco é interface SE(2) (movimentação planar), você pode experimentar comandos como "vá em frente", "vire à direita" etc. Entretanto, lembre-se que o GR00T foi treinado majoritariamente em tarefas de manipulação; comandos puramente de navegação podem não ter sido foco, mas ainda assim fornecem um input linguístico que o modelo tentará interpretar em ações.
- **Solicitando Ação ao GR00T:** Com o dict de observação montado, chame o cliente do GR00T para obter a ação. Exemplo no contexto ZMQ Python:

```
action = client.get_action(obs_dict)
```

A resposta `action` tipicamente será um array numpy ou dicionário de arrays correspondendo aos comandos para cada parte do robô (conforme definido no EmbodimentTag usado). No caso do GR1 (braços), será um vetor concatenado de comandos para juntas dos braços, mãos e cintura [30] . Se você optar por um *embodiment* diferente ou treinado para base móvel, a estrutura muda. (Nota: O repositório GR00T oferece *embodiment heads* pré-treinadas: e.g., `EmbodimentTag.GR1` para humanoide completo, `OXE_DROID` para braço único, `AGIBOT_GENIE1` para humanoide com garras, ou `NEW_EMBODIMENT` para robô novo não pré-treinado [30] . Como seu robô é custom (quadrúpede com braços), nenhuma cabeça existente corresponde exatamente. Inicialmente, use a mais próxima — possivelmente `GR1` ou `AGIBOT_GENIE1` se disponível — apenas para testar integração; para resultados ótimos, provavelmente será necessário **treinar/ajustar um novo head** para seu robô [31] .)

- **Aplicação da Ação no Isaac Lab:** Converta o output do GR00T em comandos no simulador. Por exemplo, se você recebeu um vetor de torques/velocidades para juntas, você pode aplicá-los diretamente através da API do Isaac Lab (métodos do ambiente para definir ações nas articulações). Se a intenção for **controle SE(2) da base**, você precisará mapear a saída do modelo para comandos de translação/rotação do robô:
- Caso esteja usando o GR00T sem ajuste específico, é possível que nenhum componente do vetor de ação corresponda diretamente à velocidade planar. Uma abordagem é **ignorar** as partes do vetor referentes a braços e extrair apenas algo como o movimento do “waist” ou outra parte como proxy para deslocamento, mas isso é ad-hoc.
- Uma solução robusta é **treinar o modelo para a tarefa de locomoção**: por exemplo, coletar dados de teleop (como você já tem) e *fine-tunar* o GR00T para que uma instrução "vá em frente" resulte em padrões de movimento das pernas. Isso exigiria usar `EmbodimentTag.NEW_EMBODIMENT` e treinar com demonstrações do Unitree andando. Alternativamente, se quiser controle imediato da base, pode usar um controlador simples fora do GR00T para converter comandos textuais em velocidades (ex: um mapeamento fixo: "vá em frente" = v=1 m/s, "pare" = v=0), até que o GR00T seja especializado.
- **Sincronização e Frequência:** Integrar o GR00T significa que a cada intervalo (por ex., a cada passo de controle ou a cada comando do usuário) você enviará um request e esperará a resposta. Certifique-se de que isso ocorra de forma assíncrona ou dentro de um tempo que não degrade a simulação em tempo real. Dado que uma inferência leva ~50-100ms, você pode chamá-la, por exemplo, a 10Hz e interpolar ou manter comandos nos frames entre chamadas. Use threads ou tarefas assíncronas se necessário para não travar o render/physics loop do Isaac Sim enquanto espera a IA.
- **Teste Iterativo:** Inicialmente, teste comandos simples com o robô preso no lugar (ou só com um componente, como braços) para garantir que o pipeline completo funciona: Isaac Lab -> obs -> GR00T -> ação -> Isaac Lab. Verifique que não ocorrem exceções (por exemplo, erros de dimensão no obs_dict ou na aplicação da ação). Ajuste as escalas de comando se necessário (às vezes o modelo pode retornar valores em uma escala diferente do esperado pelo controlador do robô; ex.: normalize ou escalonar velocidades).
- **Evolução para Controle Completo:** Com a integração básica funcional, planeje como unir isso ao *framework* de tarefas do Isaac Lab. Você pode criar uma nova *Task/Policy* no Isaac Lab que utiliza o GR00T. Por exemplo, ao invés de ler do teclado (teleop), ler da saída do modelo. Poderá combinar comandos: o usuário fornece uma instrução em linguagem natural, o sistema a passa ao GR00T, e o

GR00T produz o comando motor. Esse seria o *Step 5* do seu pipeline: **controle por linguagem natural assistido por IA**.

- **Fine-tuning Futuro:** Como mencionado, para que o GR00T controle eficientemente seu robô específico (especialmente a parte de locomoção), considere gerar dados simulados com o Unitree G1 (usando seu ambiente Isaac Lab) e realizar um *fine-tuning* do modelo. O GR00T fornece scripts para fine-tuning (`scripts/gr00t_finetune.py`) e suporta adicionar novos *embodiments* [31]. Com algumas demonstrações, você pode ensinar o modelo a associar comandos ("ande para frente") a movimentos apropriados de pernas. Isso solidificará a integração no longo prazo, permitindo que o GR00T atue como um verdadeiro cérebro de alto nível para seu robô tanto em simulação quanto, potencialmente, no hardware real.

## 8. Dicas de Depuração e Solução de Problemas Comuns

Mesmo seguindo os passos, é possível encontrar alguns problemas. Abaixo listamos dicas e soluções para falhas comuns:

- **Download do modelo interrompido ou lento:** Se a etapa de download do modelo travar ou corromper (por instabilidade de rede), você pode usar um gerenciador ou tentar o *git LFS*. Por exemplo: `git lfs install && git clone https://huggingface.co/nvidia/GR00T-N1.5-3B`. Isso usa *Git LFS* para baixar os arquivos grandes. Lembre de remover qualquer download parcial antes (para evitar mistura).
- **Erro na instalação do FlashAttention:** Se o comando `pip install flash-attn==2.7.1.post4` falhou, certifique-se de que você possui um compilador C++ instalado e o CUDA Toolkit compatível. Em alguns casos, pode ser necessário instalar pacotes de cabeçalhos do CUDA (ex.: `cuda-12-4` se estiver usando apt). Como alternativa, visite o repo FlashAttention e baixe um wheel pré-compilado. A recomendação da NVIDIA é usar exatamente a versão indicada e sem isolamento de build [8]. Lembre-se de ativar de novo o env correto ao tentar reinstalar.
- **Conflitos de dependências:** Seu ambiente Isaac Lab já possuía várias libs; se notar comportamentos estranhos após instalar o GR00T (por ex., versões de Torch ou NumPy conflitantes), pode ser necessário alinhar versões. O Docker do GR00T sugere usar Torch 2.5.1, NumPy 1.26, etc., reinstalando-os por segurança [32]. Caso algum componente do Isaac Lab quebre por conta disso, considere separar o GR00T em outro env e usar o modo HTTP.
- **Falha ao importar `gr00t`:** Se ao executar o servidor/cliente aparecer *ModuleNotFoundError: no module named gr00t*, significa que o pip editable install não ocorreu corretamente. Talvez você esqueceu de ativar o env ou de rodar `pip install -e .[base]`. Retorne ao passo de instalação, reinstale e verifique mensagens de sucesso (procure por *"Successfully installed gr00t"*).
- **Erro de `pyav`/FFmpeg:** Como mencionado, caso a instalação falhe em algo relacionado a `pyav` (ou ao usar funcionalidades de vídeo), instale manualmente o pacote AV (`pip install av`). Isso resolve problemas com dependência renomeada no projeto [10].
- **Porta 5555 ocupada:** Se ao iniciar o servidor nada acontece ou você sabe de outro serviço usando 5555, mude a porta (`--port` argumento em server e cliente). Evite portas reservadas ou já em uso pelo Isaac Sim (por exemplo, 9090, 8100, etc., se o Isaac Sim ou extensões as utilizam).
- **Modelo não responde (timeout):** Caso o cliente trave esperando resposta, pode ser que o servidor não recebeu a mensagem. Verifique se você especificou o host correto (no mesmo computador, use `"localhost"`). No modo ZMQ padrão, o host default é localhost mesmo [33]. Se o servidor estiver em outra máquina, passe `--host <ip>` nos dois lados.

- **Uso excessivo de CPU durante inferência:** O GR00T deve usar GPU intensivamente. Se perceber carga alta de CPU e GPU baixa, pode ser que o modelo caiu para CPU. Verifique no log do servidor se há algo como *"using device cpu"* (ou ausencia de mensagem de GPU). Isso pode ocorrer se PyTorch não encontrou a GPU ou se ocorreu um erro silencioso no FlashAttention (por ex., incompatibilidade que fez fallback). Tente forçar uso de GPU configurando `device="cuda"` na criação do `Gr00tPolicy` (se modificando código) [34] ou reinstale PyTorch com suporte CUDA.
- **Travamentos ou lentidão no Isaac Sim após integração:** Se a simulação estiver engasgando, pode ser a chamada de inferência bloqueando o loop. Verifique se você não está chamando sincronicamente em alta frequência. Use threads ou reduza a frequência. Utilize logs para medir o tempo de cada chamada `get_action` e ajuste a cadência de acordo.
- **Resultados de ação incoerentes:** Se o robô agir de forma errática aos comandos do GR00T, algumas causas podem ser: diferença no número de DOFs (modelo esperando juntas que seu robô não tem), escalas incompatíveis, ou instrução ambígua. Solução: experimente comandos diferentes, normalize inputs (por exemplo, imagens em escala de cor 0-255 vs 0-1 conforme necessário – o GR00T espera uint8 0-255 [35], então está certo enviar imagem raw 0-255). Se apenas algumas componentes do vetor importam, filtre as demais. Em último caso, vá para o fine-tuning do modelo para alinhar completamente ao seu robô.
- **Referências e ajuda adicional:** Consulte a documentação do GR00T (pasta `getting_started/` no repo) para exemplos de uso e notebooks de inferência [36]. Há também um exemplo de controle de braço robótico usando GR00T N1.5 publicado pela comunidade (Seeed Studio) que pode fornecer insights sobre integração em código [37] [38]. E lembre-se: a integração de um modelo de fundamento robótico é um processo iterativo – comece simples, valide cada parte e evolua gradualmente até o robô entender comandos em linguagem natural de forma confiável!

Finalmente, com o GR00T operando e integrado, você terá uma plataforma poderosa para controle de robôs via instruções de alto nível. Boa sorte nos experimentos, e fique atento a atualizações do projeto Isaac-GR00T para melhorias de modelo ou ferramentas de suporte. Qualquer problema específico pode ser discutido nos fóruns da NVIDIA Developer ou no repositório GitHub do Isaac-GR00T (onde já existem *issues* sobre setup e uso [39]).

**Fonte:** Este guia foi construído com base na documentação oficial do NVIDIA Isaac GR00T e experiências práticas de configuração, garantindo passos validados e referenciados para reproducibilidade. Boa exploração com seu Unitree G1 e o GR00T! [5] [40]

---

[1] [2] [3] [5] [6] [7] [11] [13] [14] [22] [23] [30] [31] [34] [36] [40] GitHub - NVIDIA/Isaac-GR00T: NVIDIA Isaac GR00T N1.5 is the world's first open foundation model for generalized humanoid robot reasoning and skills.
https://github.com/NVIDIA/Isaac-GR00T

[4] [12] [18] nvidia/GR00T-N1.5-3B · Hugging Face
https://huggingface.co/nvidia/GR00T-N1.5-3B

[8] [26] [37] [38] Control 6/7 DOF Robot Arm using NVIDIA Isaac GR00T N1.5 | Seeed Studio Wiki
https://wiki.seeedstudio.com/control_robotic_arm_via_gr00t/

[9] [25] [32] Dockerfile
https://github.com/andrearosasco/gr00tCub/blob/1731356b054b6206414b4cd12dae2ffd5f157e86/Dockerfile

10 install_gr00tn1.sh
https://github.com/isaac-for-healthcare/i4h-workflows/blob/920bccf90fd3fe765545c01696491b73ce58d0c7/tools/env_setup/install_gr00tn1.sh

15 16 17 19 20 21 24 27 28 29 33 35 inference_service.py
https://github.com/NVIDIA/Isaac-GR00T/blob/ae7d46f02cdca5d9c80efc446fe41fe2b58e94c7/scripts/inference_service.py

39 Incomplete setup instructions · Issue #3 · NVIDIA/Isaac-GR00T
https://github.com/NVIDIA/Isaac-GR00T/issues/3