ChatGPT

Lógica dos Contratos FlashLoan e Arbitragem (Poly-Flash)

O repositório **poly-flash** contém um contrato inteligente principal chamado `Flashloan.sol` que viabiliza arbitragem usando flash loans na Polygon. Este contrato foi projetado para interagir com pools de **flash loan do DODO**, aproveitando que o DODO V2 permite flash loans sem taxa. A lógica funciona em etapas:

- **Início do Flash Loan:** A função pública `dodoFlashLoan(FlashParams params)` do contrato inicia o processo [1]. Ela recebe um struct com os parâmetros: endereço do pool DODO a usar, quantidade a tomar emprestado e as rotas de swap que compõem a arbitragem. Internamente, essa função codifica os dados da operação (incluindo quem chamou, pool, quantia e rotas) e então invoca o método `flashLoan` do pool DODO correspondente [2]. O DODO então transfere instantaneamente os tokens emprestados para o contrato e chama de volta uma função de *callback* no nosso contrato.

- **Callback do Flash Loan:** O contrato `Flashloan.sol` herda de um base `DodoBase` que implementa as funções de callback esperadas pelo DODO (como `DVMFlashLoanCall`, `DPPFlashLoanCall` etc.) e todas elas direcionam para um método interno `_flashLoanCallBack` [3] [4]. Assim que os tokens emprestados caem no contrato, `_flashLoanCallBack` é executado. Nele, os dados codificados são decodificados de volta em estrutura (`FlashCallbackData`), obtendo-se o token emprestado, valor e rotas planejadas [5]. Em seguida, o contrato **valida** que recebeu a quantidade esperada do token inicial e prossegue para realizar as trocas definidas.

- **Execução das Swaps (Arbitragem):** A sequência de swaps é executada conforme as **rotas** passadas. O poly-flash suporta múltiplas rotas em paralelo (distribuindo porcentagens do valor entre rotas) e múltiplos "hops" (etapas) por rota. Cada `Route` contém um array de `Hop` descrevendo um swap em uma DEX específica, e um campo `part` indicando que porcentagem do montante total do flash loan vai por ali [6]. Primeiro, a função `routeLoop` divide o montante entre as rotas segundo os pesos (parts) [7]. Depois, para cada rota, chama `hopLoop` para iterar por cada hop (swap) daquela rota [8]. Em cada hop, a função `pickProtocol` seleciona e executa a função correta de swap de acordo com o campo `protocol` daquele hop [9]. Por exemplo, `protocol == 0` indica swap em **Uniswap V3**, `protocol < 8` indica algum **router Uniswap V2** (padrão) e protocolos ≥ 8 tratam swaps via **DODO** [10].

- *Swap Uniswap V3:* O contrato decodifica o campo `data` do hop para obter o endereço do router V3 e a *fee* de pool (por ex., 500 = 0,05%) [11]. Então aprova-se o token de entrada para o router e chama-se `exactInputSingle` passando token in, token out, fee, quantidade, etc., recebendo o token de saída [12]. O swap V3 é feito diretamente no contrato do router UniswapV3 fornecido [13].

- *Swap Uniswap V2 (Sushi/QuickSwap etc.):* Decodifica-se `data` para um endereço de router V2 [14], aprova-se o token de entrada para esse router e invoca-se `swapExactTokensForTokens` naquele router, passando a quantidade de entrada e recebendo os tokens de saída. No código do poly-flash, o retorno dessa chamada é um array de montantes para cada token na rota; eles usam o índice

`[1]` assumindo uma rota simples de dois tokens [15]. (*Obs.: Isso funciona quando o path tem apenas 2 tokens; para múltiplos hops no mesmo router V2, o último elemento do array seria o resultado final.*)

- *Swap via DODO:* Para protocolos indicados (≥ 8), eles usam a função `dodoV2Swap`. Decodifica-se `data` para obter os endereços do pool DODO, do contrato de aprovação e do proxy [16]. Em seguida, aprova-se o token de entrada ao contrato de aprovação do DODO e chama-se `IDODOProxy.dodoSwapV2TokenToToken` para realizar a troca do token inicial pelo final, dentro daquele pool DODO, passando também a direção (base→quote ou vice-versa) [17] [18].

Para todas as swaps, o contrato utiliza `SafeERC20` para aprovar e movimentar tokens com segurança, e define `amountOutMinimum` como 0 ou 1 nos swaps (ou seja, não impõe limite de slippage na execução on-chain) [13] [19] – assumindo que a pré-verificação off-chain garante o lucro, ele prioriza a execução efetiva sem bloqueios.

- **Finalização e Lucro:** Após completar todas as rotas e hops, o contrato emite um evento `SwapFinished` indicando o token inicial e o saldo resultante dele no contrato [20]. Em seguida, verifica se tem pelo menos a quantidade emprestada daquele token para devolver ao pool (caso contrário, algo deu errado e reverte a transação) [21]. Então **devolve o principal do flash loan** ao pool DODO [22]. Qualquer excedente restante do token inicial no contrato é considerado lucro – este é imediatamente transferido para o endereço que iniciou a transação (usuário/bot) [23]. Um evento `SentProfit` é emitido indicando o destinatário e quantia de lucro enviada [24] [25]. Desse modo, toda a arbitragem (empréstimo, swaps e retorno) ocorre dentro de uma única transação atômica.

Resumo: A lógica do Poly-Flash é bastante modular e robusta. Através de estruturas de rotas configuráveis, ele consegue realizar **arbitragem direta** (um hop de ida e volta entre dois DEX) ou **triangular/múltipla** (vários hops por diferentes DEX, possivelmente envolvendo 3+ tokens) usando flash loan sem capital inicial. Essa flexibilidade é evidente no script de exemplo, onde definem uma rota dividida em dois hops: primeiro USDC→WETH via protocolo 1 (UniswapV2/QuickSwap) e depois WETH→USDC via protocolo 0 (UniswapV3) [26] [27]. O design garante que, se ao final das swaps não houver fundos suficientes para pagar o empréstimo, a transação reverte (protegendo contra prejuízo) – caso contrário, paga-se o empréstimo e o lucro fica com o usuário.

Adaptação da Lógica para o Sistema Existente

O sistema existente (conforme o repositório **defi-arb**) já faz monitoramento de preços em **Uniswap V3, SushiSwap (Uniswap V2) e QuickSwap** na Polygon e detecta oportunidades de arbitragem direta e triangular [28] [29]. No entanto, a parte de execução on-chain precisa ser integrada. Atualmente, existe um contrato `FlashLoanArbitrage.sol` parcialmente implementado – ele configura o flash loan via **Aave V3** e deixa um *TODO* para a lógica de arbitragem [30]. Vamos adaptar as ideias do Poly-Flash para completar esse contrato e integrá-lo ao fluxo do bot:

- **Uso de Aave V3 para Flash Loan:** Diferente do poly-flash (que usa DODO), o contrato do sistema usa a pool do Aave V3 (endereço `0x794...a0d`) para obter flash loans [31]. Isso implica uma diferença: o Aave cobra uma pequena taxa (premium de 0.05% atualmente) sobre o valor emprestado, ao contrário do DODO que não cobrava taxa. Devemos levar isso em conta no cálculo de lucro. O contrato já possui funções `flashLoanSimple` e `executeOperation` compatíveis com a interface do Aave [32] [33]. Quando o bot quiser executar uma arbitragem:

- Ele chamará `flashLoanSimple(receiver, asset, amount, data, 0)` no contrato. Normalmente usaremos o próprio contrato como `receiver` (pois é ele que implementa `executeOperation`), e `asset` será o token a tomar emprestado.
- A pool Aave transferirá `amount` desse token para o contrato e chamará `executeOperation` no nosso contrato, passando os arrays de ativos, montantes e prêmio, além dos `params` (dados extras codificados). O contrato já verifica que quem chama é o Aave e que o `initiator` é ele mesmo [34].

- Em `executeOperation`, o contrato decodifica os `params` para obter os dados da arbitragem planejada [35] e então precisa **executar a lógica de swaps**.

- **Definição dos Parâmetros de Arbitragem:** O contrato atual define um struct simples `ArbitrageData` com `tokenA`, `tokenB`, `amount` e uma string `route` indicando as DEXs a usar [36]. Podemos continuar com esse formato para arbitragem direta (dois tokens) e representar a sequência de DEXs na string (por ex., `"uniswap->sushiswap"`). Para arbitragem triangular, precisaríamos incluir o terceiro token (ex.: WMATIC→USDC→WETH→WMATIC); como `ArbitrageData` não tem um campo para tokenC, podemos adotar uma convenção: `tokenA` será tanto o token inicial quanto o final do ciclo, `tokenB` o segundo token, e deduzir o terceiro token implícito pela combinação (ou estender o struct para incluir um array de tokens ou um `tokenC`). Uma alternativa mais robusta seria adotar uma estrutura similar à do poly-flash – por exemplo, passar uma lista de hops ou um "caminho" completo em `params`. Contudo, para manter as coisas simples, podemos codificar algumas rotações fixas baseadas na string de rota para os casos comuns.

- **Implementação das Swaps no Contrato:** Inspirados pelo poly-flash, implementaremos a função interna `_executeArbitrage(ArbitrageData data)` para realizar as trocas e calcular o lucro [37]. O pseudocódigo seria:

```
function _executeArbitrage(ArbitrageData memory data) internal returns
(uint256) {
    uint256 balanceBefore = IERC20(data.tokenA).balanceOf(address(this));
    // exemplo: data.tokenA = WMATIC, data.tokenB = USDC, data.route = "uniswap-
>sushiswap"
    if (isDirectArb(data.route)) {
        // dividir a string da rota
        (string memory dex1, string memory dex2) = splitRoute(data.route);
        // 1º swap: tokenA -> tokenB no dex1
        uint256 amountB = swap(data.tokenA, data.tokenB, data.amount, dex1);
        // 2º swap: tokenB -> tokenA no dex2
        uint256 amountA = swap(data.tokenB, data.tokenA, amountB, dex2);
    } else if (isTriangular(data.route)) {
        // parse route like "dex1->dex2->dex3"
        (string memory dex1, string memory dex2, string memory dex3) =
splitTriRoute(data.route);
        // identify tokenC (third token) via combination of tokenA/tokenB and
known tri pairs
        address tokenC = ... ;
```

```
        // swaps: tokenA -> tokenB (dex1), tokenB -> tokenC (dex2), tokenC ->
tokenA (dex3)
        uint256 amtB = swap(data.tokenA, data.tokenB, data.amount, dex1);
        uint256 amtC = swap(data.tokenB, tokenC, amtB, dex2);
        uint256 amtA = swap(tokenC, data.tokenA, amtC, dex3);
    }
    uint256 balanceAfter = IERC20(data.tokenA).balanceOf(address(this));
    return balanceAfter > balanceBefore ? balanceAfter - balanceBefore : 0;
}
```

Claro, teremos que implementar auxiliares como `swap(tokenIn, tokenOut, amount, dexName)` chamando as funções apropriadas do router correto. Podemos seguir de perto as implementações do poly-flash: - Para **Uniswap V3** (`dexName == "uniswap"` por exemplo), usar o contrato router V3 da Polygon (`UNISWAP_V3_ROUTER` já definido no contrato [38]) e chamar `exactInputSingle`. Precisaremos construir os parâmetros `ExactInputSingleParams` com `tokenIn = tokenIn`, `tokenOut = tokenOut`, `fee` adequado. *Qual fee usar?* Depende do par – muitos pares usam 0.3% (3000) ou 0.05% (500). Podemos ter isso configurado ou até mesmo calcular via dados do The Graph (há o campo `feeTier` nos pools). Uma solução simples: mapear os pares conhecidos (ex.: WMATIC/USDC usa 500, WETH/USDC usa 3000, etc.) ou passar a fee esperada no `ArbitrageData` (poderíamos acrescentar um campo, mas talvez não seja necessário se fixarmos as principais). - Para **SushiSwap e QuickSwap** (Uniswap V2 forks), usamos seus routers (já definidos: `SUSHISWAP_ROUTER`, `QUICKSWAP_ROUTER`) e chamamos `swapExactTokensForTokens`. Precisamos passar o path (por exemplo `[tokenIn, tokenOut]`), a quantidade de entrada e `amountOutMin` baixo (0 ou 1) para garantir execução. Essas funções retornam o array de outputs, do qual pegamos o último elemento como quantidade de saída obtida [15] . - Cada swap exigirá que o contrato tenha permissão de gastar o token de entrada no respectivo router. Nosso contrato já chama `_approveTokens()` no construtor para aprovar ilimitado WMATIC, USDC e WETH nos três routers principais [39] [40] . Portanto, para esses tokens não precisaremos chamar `approve` a cada vez (já otimizado). Se futuramente incluirmos outros tokens, devemos aprová-los também ou aprovar dinamicamente antes do swap (pode custar um pouco de gás extra, mas garantindo segurança).

- **Escolha do Token do Empréstimo:** Uma decisão importante ao adaptar a lógica: qual token tomar emprestado (tokenA) em cada arbitragem? No poly-flash, a rota define implicitamente isso – eles sempre pegam emprestado o token inicial da primeira rota [41] , e ao final retornam ele. No nosso sistema, a **ArbitrageService** já calcula o lucro em percentagem para cada oportunidade detectada [42] [43] , mas para executar precisamos quantificar e escolher direções. Em arbitragem direta, geralmente há duas possibilidades (ex: se WMATIC está mais barato na Uniswap do que na Sushi, podemos lucrar começando com USDC e terminando com USDC, ou vice-versa). O ideal é o sistema decidir pela direção que dá lucro:
- *Exemplo:* **WMATIC/USDC**: Uniswap preço 0.85, Sushi 0.87 USDC por WMATIC [44] . Estratégia: comprar WMATIC barato na Uniswap e vender caro na Sushi. Isso implica começar com USDC (tokenA = USDC, tokenB = WMATIC) – pegamos USDC emprestado, compramos WMATIC na Uni, vendemos WMATIC na Sushi e acabamos com mais USDC do que começamos [45] . O contrato então devolverá o flash loan em USDC e reterá o lucro em USDC.
- Se fosse o contrário (WMATIC mais barato na Sushi), faríamos tokenA = USDC, route `"sushiswap->uniswap"`.

- O bot deve determinar automaticamente essa direção. Podemos incorporar isso no cálculo de oportunidade: se a operação `tokenA -> tokenB -> tokenA` rende lucro positivo, então tokenA é o asset a pegar emprestado. A `ArbitrageService` pode simular as duas direções (já que ela tem os preços dos dois lados) e escolher a melhor.

- **Arbitragem Triangular:** O sistema detecta oportunidades triangulares (ciclos de 3 tokens) [43] . Para executá-las dentro de um único flash loan, poderíamos estender a abordagem multi-hop. Por exemplo, WMATIC→USDC→WETH→WMATIC como no exemplo [46] . Nesse caso, tokenA seria WMATIC (início/fim), tokenB poderíamos definir como USDC, e tokenC = WETH deduzido. A rota string poderia ser `"uniswap->quickswap->sushiswap"` indicando em qual DEX fazer cada conversão. Implementar genericamente exige cuidado: o contrato precisaria saber qual par trocar em cada etapa. Poderíamos:

- Determinar a ordem das trocas junto com a oportunidade encontrada (por exemplo: "WMATIC->USDC no Uni, USDC->WETH no Quick, WETH->WMATIC no Sushi").
- Passar essa informação talvez como parte de `route` (três nomes) e usar os campos `tokenA` e `tokenB` para os dois primeiros tokens, inferindo o terceiro. Contudo, inferência pode ser ambígua; talvez seja melhor modificar `ArbitrageData` para incluir todos os três tokens envolvidos numa triangular. Poderíamos, por exemplo, mudar `route` para listar também os tokens ou simplesmente passar um array de tokens no `params`.

- Uma opção inspirada no poly-flash: representar a rota completa como um array de hops dentro de `params`. Porém, isso complicaria a codificação do `params` e demandaria definir tipos complexos no contrato (structs aninhados). **Recomendação:** considerando prazos, implementar primeiro **somente arbitragem direta confiável**. Depois, pode-se evoluir o contrato para suportar rotas tri-token. De todo modo, a lógica multi-hop que escrevermos já pode suportar 3 etapas se a lista de DEX for de tamanho 3. Precisaremos apenas de uma forma de obter o token intermediário. Poderíamos manter um mapeamento estático no contrato para ciclos conhecidos (ex.: {WMATIC, USDC} -> WETH como terceiro, se sabemos que esses tokens formam triângulo com WETH nas DEX). Isso não escala para todos, mas cobre os principais.

- **Controle de Slippage e Custos:** O sistema já define thresholds de lucratividade e slippage no código off-chain [47] . É importante assegurar que, na hora da execução, a condição de lucro ainda seja válida. Como não definimos `amountOutMin` nas trocas on-chain (usamos 0 ou 1), a proteção contra slippage deve vir da lógica off-chain: o bot só deve iniciar a transação se a expectativa de lucro for superior ao threshold e descontando a taxa do flash loan e gas. Ainda assim, preços podem mudar segundos depois – então convém ser conservador no threshold para cobrir uma possível pequena variação. Poderíamos, para segurança extra, comparar o saldo final obtido no contrato com o inicial antes de reembolsar e, se o lucro for menor que 0 (ou abaixo de algum mínimo absoluto), abortar. Porém, abortar (revert) significaria pagar apenas as fees de gas, já que flash loan não seria pago se revertido; isso evita perda financeira mas ainda custa a transação. Muitos bots preferem executar somente se margem folgada. Portanto, a recomendação é calibrar bem `minProfitabilityThreshold` (ex.: 0.5% já configurado) e talvez incluir um pequeno buffer para cobrir o 0.05% da Aave e o 0.3% de cada swap V2 (cada swap V2 consome ~0.3%, V3 consome conforme fee tier). No código, podemos calcular aproximadamente o impacto das taxas: p.ex., em arbitragem direta envolvendo Sushi (0.3%) e Uni v3 (0.05%), o custo percentual é ~0.35% + flash loan

0.05% = ~0.4%. Então um spread de 0.5% mal cobre os custos – talvez usar 1% como mínimo fosse mais seguro para profit real. Essas considerações devem refletir na configuração do sistema.

- **Retorno e Gestão do Lucro:** Diferentemente do poly-flash que enviava o lucro diretamente ao EOA, nosso contrato armazena os lucros acumulados por token em um mapeamento (`profits`) e permite saque pelo owner [48] [49]. Isso é bom por segurança, para que só a conta controladora retire fundos. Após executar `_executeArbitrage`, nosso contrato adiciona o lucro obtido (em tokenA) ao registro [50]. Podemos manter assim. O bot, rodando com a chave do owner, poderá eventualmente chamar `withdrawProfit(token, amount, dest)` para transferir para a carteira ou reinvestir. Apenas certifique-se de que `executeOperation` retorna true para sinalizar sucesso ao Aave (o contrato já faz isso [51]). Também não esqueça de aprovar o retorno do empréstimo + taxa para o Aave no final de `executeOperation` – o código já faz: `IERC20(assets[0]).approve(AAVE_POOL, amounts[0] + premiums[0])` [51], permitindo que a pool retire os tokens de volta.

Em resumo, a adaptação envolve **incorporar a lógica de múltiplos swaps dentro do nosso contrato**, usando como modelo as funções do poly-flash. Precisaremos incluir interfaces para os routers (pode-se copiar `IUniswapV2Router` e `ISwapRouter` do poly-flash ou instalar via NPM pacotes da Uniswap). Assim, o contrato FlashLoanArbitrage conseguirá por si só trocar tokens em Uniswap, Sushi e QuickSwap durante o callback do flash loan. Após implementar e testar cuidadosamente essa parte, o sistema existente terá a "camada de execução" completa: ao detectar uma oportunidade, chamará o contrato para executar a arbitragem automaticamente.

Integração dos Scripts Hardhat (Deploy & Execução) ⚙

Para unificar os fluxos, também vamos integrar os scripts Hardhat do poly-flash (ajustados) no contexto do nosso projeto. Atualmente, no **defi-arb** temos um script `scripts/deploy.js` que faz o deploy do contrato `FlashLoanArbitrage` na rede desejada [52], verifica no Polygonscan e salva o endereço em um arquivo JSON de deployments [53] [54]. Esse script deve ser revisitado após modificarmos o contrato: - **Deploy do Contrato:** Se renomearmos ou adicionarmos algum construtor no contrato, atualizar o script de deploy. No momento, o contrato não exige parâmetros no construtor, então o deploy é simples. Devemos manter a prática de **salvar o endereço** em um arquivo (ou variável de ambiente) para uso posterior. O script atual já sugere inserir o endereço no `.env` (`FLASH_LOAN_CONTRACT_ADDRESS`) [55] – é uma boa ideia seguir isso, para que o bot possa ler de config.

- **Configuração no Sistema:** No arquivo de configuração central (`src/config.js` talvez), incluir o endereço do contrato deployado. Além disso, precisamos garantir que as credenciais (RPC URL, PRIVATE_KEY) estejam configuradas para permitir o deploy e as interações on-chain [56]. Já usamos a Alchemy RPC e a private key no .env para Hardhat.

- **Script de Execução/Chamadas:** O poly-flash provê um exemplo de script TypeScript (`scripts/flashloan.ts`) que demonstra como invocar o contrato flash loan para executar uma arbitragem específica [57] [58]. Podemos criar algo similar em nosso projeto (em JavaScript/TypeScript):

- Esse script receberia ou definiria internamente os parâmetros de um arbitragem (por exemplo, poderíamos fornecer via linha de comando o par e as DEXs a usar, ou pegar do nosso serviço de arbitragem).
- Então conecta no contrato já deployado (usando o `FLASH_LOAN_CONTRACT_ADDRESS` e ABI do contrato).
- Monta os dados `ArbitrageData` e codifica para bytes. Em JS com ethers, podemos usar `ethers.utils.defaultAbiCoder` para codificar o struct como tupla. Por exemplo:

```
const abiCoder = new ethers.utils.AbiCoder();
const arbitrageData = [tokenA, tokenB, amount, routeString];
const paramsData = abiCoder.encode(
   ["tuple(address tokenA, address tokenB, uint256 amount, string route)"],
   [arbitrageData]
);
await flashLoanContract.flashLoanSimple(flashLoanContract.address, tokenA,
amount, paramsData, 0);
```

Obs.: Note que passamos `flashLoanContract.address` como `receiver` para que a própria instância receba o flash loan e execute o callback. `tokenA` (asset) é o token emprestado. `amount` deve estar em base de Wei (usar `ethers.utils.parseUnits` conforme decimal do token). `routeString` seria algo como `"uniswap->sushiswap"`.

- Opcionalmente, podemos especificar opções de gas. A Hardhat com fork local ignora isso, mas em rede de verdade talvez definamos um gasLimit alto (p.ex 2-3 milhões) e um gasPrice adequado. O exemplo do poly-flash usava 300 gwei na Polygon (um valor muito alto, fora do normal) apenas ilustrativo [59]. Podemos usar nosso config de gas (por ex., 50 gwei max, 30 gwei priority como sugerido no TECHNICAL.md [60]) ao enviar a tx.

- Após enviar a transação, podemos imprimir o hash e eventualmente aguardar confirmação e verificar o resultado (saldo final do contrato ou evento emitido).

Esse script/manual serve para testes. **Entretanto**, no fluxo real do bot, não precisaremos rodar um script manual – o próprio `BlockchainService` do nosso projeto deve chamar o contrato quando a ArbitrageService sinalizar uma oportunidade: - No `blockchainService.js`, podemos integrar o contrato via ethers usando o ABI gerado pelo Hardhat (ou podemos importar o ABI JSON gerado na compilação Hardhat para o projeto Node). Com isso, chamamos `flashLoanSimple` programaticamente. Por exemplo:

```
const flashLoanContract = new ethers.Contract(contractAddress, FlashLoanABI,
signer);
await flashLoanContract.flashLoanSimple(contractAddress, asset, amount,
paramsData, 0);
```

Esse chamado disparará a tx na Polygon. Devemos envolver em try/catch para capturar eventuais erros (ex.: se a transação reverter por slippage, out of gas, etc). Também podemos usar `estimateGas` antes de enviar para ter certeza que o gasLimit está suficiente e para fins de logging de custo.

- Importante: **Tempo de confirmação.** O bot deve idealmente enviar a transação e não esperar bloqueantemente (pode só logar e seguir monitorando, ou aguardar a Promise resolver). Como arbitragem é sensível a timing, possivelmente usar `sendTransaction` sem esperar ou com baixa confirmação (Polygon geralmente finaliza rápido ~2s). Isso é uma decisão de implementação de bot.

- **Hardhat Tasks:** Podemos adicionar tasks no `hardhat.config.ts` para facilitar operações. Por exemplo, uma task `npx hardhat arb --pair WMATIC/USDC --route uniswap->sushiswap --amount 1000` que internamente chama a função acima. Isso pode auxiliar em testes manuais na rede fork ou mesmo mainnet (com muito cuidado). O poly-flash tinha comandos yarn para testar e deployar (veja `package.json` deles).

- **Docker/CI:** Se estamos usando o Docker ou pipelines, após integrar o contrato e scripts, lembrar de atualizar o fluxo de build/execução. O poly-flash-bot original possuía um Docker config e CI badge, então talvez incorporar que nosso sistema agora depende do contrato deployado. Podemos até rodar um `deploy.js` automático na inicialização em fork (rede local) para garantir que há um contrato para usar nos testes.

Em suma, a integração dos scripts Hardhat garante que temos **ferramentas para implantar e acionar** o contrato flashloan facilmente. No desenvolvimento: - Primeiro rodamos `npx hardhat run scripts/deploy.js --network polygon` para fazer o deploy na mainnet (ou `--network mumbai` para testes). - Atualizamos o config com o endereço. - Depois, para testar num fork, podemos rodar `npx hardhat run scripts/flashloan.js --network hardhat` simulando uma arbitragem (ou escrever testes automatizados no Mocha/Chai como o poly-flash fez). Por exemplo, podemos criar um teste que faz flashloan de 1000 USDC e executa USDC->WETH->USDC via Uniswap e checa se o contrato teve lucro > 0, similar ao teste do poly-flash [61] [62]. Adaptações serão necessárias para usar Aave no lugar de DODO (provavelmente teremos que impersonar o Aave pool ou usar Hardhat mainnet fork para que o pool funcione real). - Integrar esses testes ao nosso fluxo dá confiança antes de colocar em produção.

Ambiente de Teste com Hardhat Fork (Polygon)

Testar essa integração em um ambiente controlado é fundamental antes de arriscar fundos reais. Vamos usar o Hardhat **Mainnet Fork** da Polygon para simular cenários de arbitragem. Nosso `hardhat.config.js` já está preparado para forkar a Polygon usando Alchemy e até fixa um número de bloco (50.000.000) [63]. Os passos recomendados:

1. **Iniciar Hardhat Node com Fork:** Rode `npx hardhat node` (por padrão ele já utilizará a config de fork da Polygon). Isso iniciará um nó local HTTP://127.0.0.1:8545 que simula a rede Polygon naquele bloco específico. Podemos ajustar o `blockNumber` para um bloco onde sabemos que havia uma oportunidade. Por exemplo, o poly-flash menciona uma tx de exemplo no bloco 22.593.589 [64] – poderíamos usar esse ou procurar nos gráficos de preços um momento de desalinhamento. Alterne o `hardhat.fork.blockNumber` conforme desejado.

2. **Impersonar Contas e Ajustar Estado (se necessário):** No fork, podemos **impersonar** contas para manipular liquidez ou saldo. Por exemplo, poderíamos:

3. Dar fundos à nossa conta do bot: usando `hardhat_impersonateAccount` para uma whale de MATIC e transferir MATIC para pagar gas, ou diretamente configurar `PRIVATE_KEY` com algum valor de teste se for faucet.

4. Impersonar um LP e skew o preço em um pool para criar manualmente arbitragem: ex., pegar o contrato de um pool Uniswap e ajustar reservas via swaps forçados. Isso é avançado, mas possível. Mais fácil é: se queremos simular uma arbitragem real, podemos não mexer no estado e apenas identificar uma que já exista naquele estado do fork.

5. **Detectar Oportunidade no Fork:** Podemos rodar nosso bot em modo dev conectado ao fork local. Configure o RPC do bot para `http://localhost:8545` e rode `npm run dev`. O GraphService continuará puxando dados reais do TheGraph (que reflete o estado próximo ao bloco atual, não do fork fixo, mas podemos configurar TheGraph para pegar dados históricos também se preciso). Como simplificação, podemos usar o fork com blocos recentes e supor que oportunidades semelhantes possam ser encontradas. Ao rodar o bot contra o fork, quando ele detectar uma oportunidade, ele chamará o contrato. Essa chamada ocorrerá no nó local (fork), executando nosso código de arbitragem contra o estado clonado da mainnet.

6. **Simular Execução Manual:** Alternativamente, use uma script/test manual no fork: por exemplo, invocar uma arbitragem conhecida. Podemos tentar reproduzir a situação de lucro do poly-flash:

7. Exemplo: No fork, pegar o pool DODO WETH/USDC e Uniswap/Apeswap combos citados nos logs [65] [66]. Porém, nosso contrato usa Aave, então melhor focar em Uniswap/Sushi. Podemos identificar um par com spread no momento (0.5%+) e tentar uma pequena quantia.

8. Executar via `flashLoanContract.flashLoanSimple(...)` e observar os eventos `ArbitrageExecuted` ou o saldo do contrato. Em ambiente fork, podemos consultar depois `profits` no contrato ou fazer `owner.withdrawProfit` para ver quanto ganhou, sem risco real.

9. **Verificações Pós-Tx:** No output do Hardhat node veremos os logs de eventos. Nosso contrato emite `FlashLoanExecuted` quando pega o empréstimo, `ArbitrageExecuted` quando finaliza a arbitragem (com lucro calculado) e possivelmente nossos eventos internos de debug (podemos adicionar alguns ou usar console.log no Hardhat contrapartida). Por exemplo, esperamos ver `ArbitrageExecuted(tokenA, tokenB, profit, route)` no caso de sucesso [48]. Podemos também verificar que `profits[tokenA]` no contrato aumentou do valor anterior.

10. **Testes Automatizados:** Como parte do desenvolvimento, escrever testes unitários focados nas funções do contrato pode ajudar:

11. Um teste para arbitragem direta: config uma situação (talvez usando impersonation para fornecer tokens ao contrato para não reverter em caso de pequena perda) e checar se `_executeArbitrage` consegue executar swaps e resultar no saldo esperado. Por exemplo, similar ao teste do poly-flash que verifica que após chamar flashloan USDC->WETH->USDC o saldo de USDC

do usuário (owner) aumentou [62] [67]. No nosso caso, como o lucro fica no contrato, podemos verificar `profits[USDC] > 0`.

12. Teste de não-lucro: cenários em que não há spread suficiente devem, idealmente, não ser executados pelo bot. Mas se executados, o contrato poderia terminar com zero lucro ou até falhar no require de pagamento (o que reverteria a tx). Simular isso nos testes garante que o require de segurança está funcionando.

13. **Ajuste Fino:** O ambiente de fork permite experimentar sem custos, então é valioso testar com várias combinações de tokens e DEX. Por exemplo:

14. WMATIC/USDC (Uni vs Sushi),
15. USDC/DAI (Uni v3 vs Curve, embora Curve não esteja integrado – mas poderíamos integrar 1inch via DODO or etc se expandido),
16. WETH/USDC (Sushi vs QuickSwap).

17. Ciclos triangulares populares: WMATIC-USDC-WETH, DAI-USDC-USDT (stable triangle, geralmente sem lucro mas testar overhead). Observando os resultados, conseguimos calibrar parâmetros do bot (thresholds) antes do deploy real.

18. **Testnet (Opcional):** Além do fork, podemos testar em rede de teste (Mumbai) implantando o contrato lá e criando pools artificiais com diferenças de preço. Mumbai tem Aave e Uniswap/Sushi deployments? Aave sim, Uniswap possivelmente, Sushi não certeza. O esforço talvez não compense. O fork tende a ser mais fiel.

Concluindo, o Hardhat fork nos dá um **sandbox fiel** para validar que nossa integração funciona: desde o chamado do flash loan, passando pelas swaps, até o reembolso e cálculo de lucro, tudo automático. Qualquer erro (p.ex., falta de aprovação, endereço errado de router, ou lógica incorreta nas trocas) aparecerá aqui primeiro. Somente após os testes serem bem-sucedidos deveremos implantar o contrato na Polygon mainnet e habilitar o bot para rodar com ele em produção.

Prompt de Integração e Unificação dos Repositórios (RooCode)

Por fim, vamos organizar os passos de integração de forma consolidada – um tipo de *checklist* que poderia ser usado no **RooCode** (ou outro assistente) para aplicar as mudanças de forma consistente unindo o repositório poly-flash ao nosso. Os principais pontos de integração são:

1. **Adicionar Dependências e Interfaces Necessárias:** Incorpore no repositório do bot as interfaces de contratos usadas no poly-flash:
2. Criar arquivo `contracts/interfaces/IUniswapV2Router.sol` contendo a interface mínima do router Uniswap V2 (funções `swapExactTokensForTokens(uint,uint,address[],address,uint)` etc.). Pode copiar do poly-flash ou do repositório Uniswap.
3. Adicionar `contracts/uniswap/v3/ISwapRouter.sol` interface do Uniswap V3 router (contendo `exactInputSingle` e talvez `exactInput` se precisar). Poly-flash já tinha esse arquivo [68].
4. Se decidirmos eventualmente usar DODO (talvez não agora), interfaces `IDODO.sol` e `IDODOProxy.sol` estão no poly-flash e poderiam ser adicionadas.

5. Certifique que `package.json` inclua `@openzeppelin/contracts` (já usamos), e possivelmente instale `@uniswap/v3-periphery` npm package para reutilizar tipos se preferir (opcional, interface manual é ok).

6. Adicionar a biblioteca SafeERC20 (já usamos OpenZeppelin, só import no contrato) e, se necessário, SafeMath (Solidity 0.8 não necessita, mas poly-flash usou SafeMath por legado).

7. **Incorporar Lógica de Swaps no Contrato:** Editar o arquivo `FlashLoanArbitrage.sol`:

8. No topo, adicionar `import` das interfaces dos routers Uniswap/Sushi/Quick que incluímos.
9. Dentro de `_executeArbitrage`, substituir o trecho *TODO* por código que realiza as trocas conforme explicado. Por exemplo, implementar um `if/else` para diferentes comprimentos de rota (2 ou 3 DEX) ou até um loop genérico se convertemos a string de rota em array. Dada a simplicidade, um if para `"X->Y"` vs `"X->Y->Z"` pode ser suficiente.
10. Implementar funções auxiliares como `swapUniswapV3(tokenIn, tokenOut, amount, fee)` e `swapUniswapV2(routerAddress, tokenIn, tokenOut, amount)` ou similar, para deixar `_executeArbitrage` claro. Podemos hardcode os endereços dos routers usando as constantes globais já definidas (por exemplo, `if (dex == "uniswap") use UNISWAP_V3_ROUTER, if "sushiswap" use SUSHISWAP_ROUTER, etc.`). Como alternativa, um pequeno mapping de string->address, mas em Solidity isso exigiria `bytes32` keys ou similar; pode ser overkill – strings comparadas diretamente são caras. Talvez melhor usar enums ou códigos: ex.: definir `enum Dex { UNISWAP_V3, SUSHISWAP, QUICKSWAP }` e em vez de string passar um array de `Dex`. Contudo, isso envolveria mudar ArbitrageData.route de string para, digamos, array of uint8. Se quisermos minimizar alterações no struct, podemos interpretar a string no próprio front (BlockchainService) e passar já um formato codificado. *Solução simples:* usar a string no contrato apenas para emitir evento e log, mas internamente decidir por parâmetros: Podemos, por exemplo, acrescentar campos `uint8 dexA` e `uint8 dexB` no ArbitrageData (0=Uni,1=Sushi,2=Quick) – mas isso exigiria redeploy do contrato e mudança na codificação. Uma alternativa sem mudar o struct: codificar a sequência de dex dentro do `params` bytes separadamente e ignorar a string para lógica interna. No espírito do prompt, poderíamos instruir o RooCode a simplesmente **decodificar a string**: ainda que não elegante, é factível. Ex.: `if (keccak256(bytes(route)) == keccak256("uniswap->sushiswap")) { ... }`. Para poucas combinações é viável, embora não escalável. Dado o escopo, podemos implementar essas condições explícitas para Uniswap/Sushi, Sushi/Uniswap, Uniswap/Quick, Quick/Uniswap, Sushi/Quick, Quick/Sushi (as 6 permutações diretas), e talvez 6 para triangulares mais comuns. Isso cobre o necessário e podemos refatorar depois.
11. Garantir que após as swaps, calculamos o lucro corretamente: `profit = balanceAfter - balanceBefore`. Lembrando que no caso de triangular tokenA = token inicial/final, isso funciona. No caso de direta, tokenA inicial e final também deverá ser o mesmo para profit – escolhemos tokenA assim propositalmente (o que for emprestado é inicial e final).
12. Emitir o evento `ArbitrageExecuted` já está no código [48]. Certifique que os valores estão corretos (tokenA, tokenB, profit, route).

13. Manter a aprovação para Aave e retornar true (já feito).

14. **Unificar Configurações e Constantes:** Verifique se os endereços das DEX constantes no contrato estão atualizados:

15. `UNISWAP_V3_ROUTER = 0xE592...564` (router oficial Uniswap v3 Polygon, ok),
16. `SUSHISWAP_ROUTER = 0x1b02...506` (router SushiSwap Polygon, ok),
17. `QUICKSWAP_ROUTER = 0xa5E0...8ff` (router QuickSwap v2, ok). Esses já estão definidos [38]. Caso queiramos suportar QuickSwap V3 (que também existe na Polygon com outro endereço), poderíamos adicionar, mas não foi pedido explicitamente. Provavelmente monitoramos QuickSwap v3 via Graph, mas executar via v2? Essa discrepância é algo a esclarecer – QuickSwap v3 lançada separadamente. Supondo que as pools monitoradas incluam Quick v2, estamos coerentes.
18. Aave pool address: já definido como constante AAVE_POOL [31], certifique-se que é o v3 correto (0x794...b4814aD). Parece ser.

19. Tokens principais: WMATIC, USDC, WETH constantes [69]. Se quisermos arbitragear outros tokens, teremos que aprová-los manualmente (ou poderíamos armazenar todos tokens monitorados no contrato – não ideal). Alternativa: modificar `_approveTokens()` para aprovar qualquer token que vier a ser usado dinamicamente antes de swap, usando a mesma lógica do poly-flash `approveToken(token, router, amt)` [70]. Mas isso consome gas em runtime. Como nosso foco inicial são esses tokens base, manteremos pré-aprovação neles. Se precisar acrescentar, podemos adicionar alguns conhecidos (USDT, DAI talvez) e atualizar o function `_approveTokens` e as constantes.

20. **Mesclar Estrutura de Projeto:** O poly-flash era um projeto Hardhat TS completo. Nosso projeto já tem Hardhat configurado, então em vez de mesclar tudo, traremos apenas o necessário:

21. Coloque os testes relevantes do poly-flash (adaptados) sob `test/` no nosso repo. Ex.: criar `test/PolygonFlashloan.test.js` (ou .ts) que simule uma arbitragem direta usando nosso contrato com Hardhat fork. Podemos inspirar-nos no conteúdo de `dodoflash.test.ts` e `flashloan.test.ts` do polyflash [61] [71], removendo a parte de DODO e ajustando para Aave. Isso garante que não quebramos nada ao integrar.
22. Trazer eventuais utils úteis: poly-flash tinha utilidades em `utils/` (funções de getBigNumber, impersonate, etc. [72]). Se forem úteis nos testes, podemos incorporar.

23. Unificar README/documentações: Atualize o README do nosso projeto para mencionar que é necessário deployar o contrato e configurar o endereço, etc. Podemos também adicionar referência ao wiki do poly-flash ou posts (como o autor do poly-flash tem artigos sobre flashloans em Polygon [73]).

24. **Integração com BlockchainService:** Ajustar o código do `src/services/blockchainService.js` para usar o novo contrato:

25. Carregar o ABI JSON do FlashLoanArbitrage (pode ser gerado pelo Hardhat ou escrito manual minimal).
26. Instanciar o contrato via ethers usando signer do bot.
27. Adicionar método para executar arbitragem: algo como `async function executeArbitrageOpportunity(opportunity)`. Este método receberia os detalhes da oportunidade (por ex., tokens e DEXs a usar, e quantidade ótima calculada). Construir o `ArbitrageData` e codificá-lo. Em seguida, chamar o flashLoan.
28. Já vimos no TECHNICAL.md que eles prepararam até o ABI do Aave pool para possivelmente chamar flashLoan direto [74], mas agora chamaremos nosso contrato, o que é mais simples. Podemos

remover o uso direto do ABI do Aave no BlockchainService, pois delegaremos ao contrato on-chain essa parte.
29. Implementar logs úteis: exibir tx hash, uso de gas, evento de lucro.
30. Considerar usar `provider.estimateGas` antes da tx para pegar custo (no TECHNICAL.md tem exemplo [75]). Isso pode ajudar a decidir se vale a pena (se gasCost > expectedProfit, talvez abortar).

31. Garantir manejo de erros: se a transação falhar (reverter), capturar e logar, mas não deixar o bot travar. Ele deve continuar monitorando outras ops. Talvez implementar um contador de falhas e se muitas falharem, reavaliar parâmetros.

32. **Teste de Integração End-to-End:** Após tudo implementado, rodar o sistema completo contra o Hardhat fork local em modo dev contínuo. Ele deve detectar uma arbitragem e executar a tx no fork. Observar no console do bot e no console do Hardhat node se tudo funcionou (lucro calculado e transferido). Ajustes finais podem ser necessários aqui (por ex., formatação de dados, ou talvez vimos que faltou aprovar algum token).

33. **Preparação para Produção:** Quando satisfeito, fazemos o deploy final na mainnet:

34. Executar `scripts/deploy.js --network polygon` e anotar o endereço do contrato. Colocar no .env/config.
35. Verificar saldo de MATIC da conta deployer e também do bot para gas.
36. Iniciar o bot normalmente (`npm start` apontando para mainnet RPC). Monitorar logs de oportunidades e execuções reais. Recomenda-se começar com limites baixos (p.ex., não habilitar todas as oportunidades, ou colocar um parâmetro de "dry-run" onde ele detecta e loga sem realmente executar, para ver quão frequentes são e se as calculadas realmente seriam lucrativas). Depois, habilitar execuções reais gradualmente.

37. Ter alertas (o sistema já tem integração de NOTIFICATIONS.md provavelmente para avisar lucros ou erros).

38. **Manutenção:** Documentar no repositório as alterações. Atualizar o `TECHNICAL.md` para descrever a lógica on-chain implementada (por exemplo, especificar que agora `_executeArbitrage` realiza as swaps entre DEX usando UniswapV3/Sushi/etc, e listar as suportadas) para futuros desenvolvedores entenderem.

Este **prompt de integração** resume o plano para unir as capacidades do poly-flash (contrato inteligente de arbitragem via flash loan) com nosso sistema de monitoramento. Seguindo esses passos, teremos uma solução consistente: o monitor off-chain identifica e aciona, e o contrato on-chain executa e garante o resultado financeiro, tudo de forma otimizada e segura na rede Polygon. Boa integração e bons lucros!

[1] [30]

---

1 2 5 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 68 70 Flashloan.sol
https://github.com/yuichiroaoki/poly-flash/blob/bdc7be2c2597c544d17241d6acf2bc22d020af2d/contracts/Flashloan.sol

3 4 41 DodoBase.sol
https://github.com/yuichiroaoki/poly-flash/blob/bdc7be2c2597c544d17241d6acf2bc22d020af2d/contracts/base/DodoBase.sol

6 IFlashloan.sol
https://github.com/yuichiroaoki/poly-flash/blob/bdc7be2c2597c544d17241d6acf2bc22d020af2d/contracts/interfaces/IFlashloan.sol

26 27 57 58 59 flashloan.ts
https://github.com/yuichiroaoki/poly-flash/blob/bdc7be2c2597c544d17241d6acf2bc22d020af2d/scripts/flashloan.ts

28 29 56 README.md
https://github.com/oguidomingos/defi-arb/blob/806be20e79ffd87beb0d5ff7ed6b549d37689deb/README.md

30 31 32 33 34 35 36 37 38 39 40 48 49 50 51 69 FlashLoanArbitrage.sol
https://github.com/oguidomingos/defi-arb/blob/806be20e79ffd87beb0d5ff7ed6b549d37689deb/contracts/FlashLoanArbitrage.sol

42 43 44 45 46 47 60 74 75 TECHNICAL.md
https://github.com/oguidomingos/defi-arb/blob/806be20e79ffd87beb0d5ff7ed6b549d37689deb/TECHNICAL.md

52 53 54 55 deploy.js
https://github.com/oguidomingos/defi-arb/blob/806be20e79ffd87beb0d5ff7ed6b549d37689deb/scripts/deploy.js

61 62 67 71 72 flashloan.test.ts
https://github.com/yuichiroaoki/poly-flash/blob/bdc7be2c2597c544d17241d6acf2bc22d020af2d/test/polygon/uniswap/flashloan.test.ts

63 hardhat.config.js
https://github.com/oguidomingos/defi-arb/blob/806be20e79ffd87beb0d5ff7ed6b549d37689deb/hardhat.config.js

64 65 66 README.md
https://github.com/yuichiroaoki/poly-flashloan-bot/blob/4590e267bcb0b4ad8628de283e4fc035ec0693e7/README.md

73 GitHub - yuichiroaoki/poly-flash: Flashloan on Polygon
https://github.com/yuichiroaoki/poly-flash