ChatGPT

Reestruturação e Plano de Implementação do Projeto APLAT Quente

Análise Detalhada do Repositório Atual

Objetivo e Escopo do Projeto

O repositório **aplatquente** tem como propósito automatizar, via Python (Selenium + Microsoft Edge), todo o fluxo de geração e preenchimento de uma **Permissão de Trabalho a Quente** no sistema APLAT da plataforma P-18 [1]. Em outras palavras, o script **gera automaticamente o plano de trabalho a quente** (a partir da descrição e características da etapa) e **preenche os principais formulários no APLAT** de forma padronizada [1]. Os formulários abrangidos pela automação incluem:

- **Questionário PT** (Perguntas de Permissão de Trabalho);
- **EPI adicional necessário e proteções**;
- **Análise Ambiental**;
- **APN-1** (Análise Preliminar de Risco específica);
- **Aba de EPIs por categoria** (tabela de EPIs obrigatórios por tipo) [1].

O objetivo principal é **reduzir o trabalho manual** ao preencher essas informações repetitivas e **padronizar as respostas** conforme regras fixas e gatilhos definidos a partir do conteúdo da *Descrição da Etapa* e das *Características do Trabalho* [2]. Em resumo, dado que uma etapa de trabalho possui descrição livre e marcações de características (ex.: trabalho em altura, espaço confinado, solda, etc.), o programa deduz quais respostas se aplicam e preenche tudo automaticamente, garantindo consistência.

Importante: trata-se de um projeto específico para o ambiente corporativo Petrobras (sistema APLAT na unidade P-18) [2]. Caso seja usado em outro local ou se o sistema APLAT for atualizado, pode ser necessário ajustar *XPaths*, textos e regras para manter a compatibilidade [3].

Arquitetura e Fluxo da Solução Atual

O projeto está estruturado em módulos Python separados, cada um responsável por uma parte do fluxo. O script principal `aplatquente.py` orquestra tudo, enquanto módulos auxiliares (`quent1_infra.py`, `quent2_plano.py`, etc.) implementam funcionalidades específicas. Em alto nível, o fluxo de execução é o seguinte [4] [5]:

1. **Login no APLAT:** O script inicia abrindo o Microsoft Edge (via WebDriver `msedgedriver.exe`) e acessando a URL principal do APLAT. Tenta realizar login automaticamente (`attempt_auto_login`), utilizando credenciais do *keyring* se disponíveis (`--use-keyring`), ou então aguarda o usuário efetuar login manualmente se necessário [6]. Após o login, espera a página carregar completamente antes de prosseguir [6].

2. **Pesquisa e abertura da etapa:** Para cada número de etapa informado (parâmetro `--valor`), o script navega até a tela de pesquisa de PT, insere a data (`--data`) e o número da etapa, e

clica em “Pesquisar”. Em seguida, abre o primeiro resultado encontrado, carregando os dados da PT selecionada [7]. Nesse ponto, a tela de **Dados da Etapa** da PT específica está aberta.

3. **Validação do tipo "Trabalho a Quente":** Com a etapa aberta, o programa verifica se o **Tipo de Trabalho** selecionado é exatamente "Trabalho a Quente". Essa verificação é crucial para garantir que só se aplique o preenchimento automatizado nas PTs corretas. Se o tipo de trabalho não for *Trabalho a Quente*, o script emite um aviso e pula aquela etapa, prosseguindo para a próxima (se houver) [5]. Caso seja *Trabalho a Quente*, a automação continua.

4. **Coleta de contexto (Descrição e Características):** Em seguida, o script coleta o texto da **Descrição da Etapa** e as **Características do trabalho** marcadas. Essa coleta é feita de forma resiliente, tentando múltiplos seletores/XPath para encontrar as informações, já que a estrutura HTML pode variar. Por exemplo, para características, o código tenta primeiro encontrar todos os elementos `<span>` com classe indicativa, depois tenta localizar um `<fieldset>` específico, e até recorre a regex no texto completo como *fallback* [8] [9]. O texto coletado é então **normalizado** (caixa alta, sem acentos, espaçamento uniforme) para facilitar comparações e geração de regras [10].

5. **Geração do plano de preenchimento:** Com base na descrição e características normalizadas, o módulo de plano (`quent2_plano.py`) monta um **contexto de flags booleanas** indicando a presença de certos riscos ou condições. Por exemplo, flags como `tem_chama`, `tem_solda`, `tem_altura`, `tem_espaco_confinado`, entre outras, são definidas como **True/False** dependendo se palavras-chave associadas aparecem no texto [11]. A partir dessas flags, o programa gera:

6. As respostas para o **Questionário PT** (cada pergunta será marcada como "Sim", "Não" ou "N/A" conforme as regras);
7. As marcações de **EPI adicional necessário** (radios de EPI que devem ser "Sim" ou "Não");
8. A seleção de EPIs por categoria na tabela da aba EPI (por exemplo, incluir *luva de raspa* se há trabalho com chama, etc.);
9. As respostas *Sim/Não* do formulário **APN-1** (até Q020) indicando se há certos riscos presentes;
10. As respostas do formulário **Análise Ambiental** (no caso de trabalho a quente, tipicamente todas "Não").

Esses valores são montados combinando bases padrão de respostas com ajustes condicionais. O código define dicionários base (por exemplo, `QPT_BASE`, `EPI_RADIOS_BASE`, etc.) com respostas default e depois os modifica de acordo com as flags de contexto [12] [13]. Assim garante-se que, por exemplo, se `tem_altura=True`, a resposta da pergunta de cinto de segurança em EPI adicional passe de "Não" para "Sim" [13], ou que certos EPIs obrigatórios sejam adicionados. Ao final dessa etapa, o programa imprime no console um **relatório detalhado do plano** gerado, mostrando tudo que será preenchido em cada aba [14] – isso ajuda a revisar as decisões antes de efetivamente gravar.

1. **Preenchimento automático das abas:** Após montar o plano, o script começa a navegar pelas abas da PT e preencher cada formulário de acordo com o plano. A sequência típica é:
2. **Questionário PT:** navega até a aba e preenche cada pergunta marcando Sim/Não/NA conforme o mapeamento gerado [15].
3. **EPI Adicional Necessário:** preenche os campos de EPI adicional (geralmente um conjunto de perguntas de sim/não sobre necessidade de certos EPIs ou proteções adicionais) [15].
4. **Análise Ambiental:** marca todas as respostas (no caso de trabalho a quente, tipicamente "Não" para todos) [14].

5. **APN-1:** preenche as respostas Sim/Não até a questão Q020 conforme o plano de riscos identificado [14].
6. **Tabela de EPIs por categoria (Aba EPI):** insere os EPIs necessários em cada categoria (vestimentas, luvas, etc.), comparando o que já está selecionado com o que deveria estar, e adicionando se algo faltar. Esse passo utiliza rotinas específicas para lidar com a tabela dinâmica de EPIs e garantir que cada item necessário esteja marcado [16].

Para cada formulário, o código usa funções de apoio em `quent3_preenchimento.py` e `quent4_epi.py` que localizam elementos por texto ou código e tentam interagir de forma robusta. Por exemplo, para clicar em opções de rádio, há tentativas graduais: primeiro clicar diretamente no elemento `<input>`, se falhar tenta clicar no `<label>` associado, se ainda falhar executa JavaScript `arguments[0].click()` [17]. Isso torna o preenchimento mais resiliente a pequenos problemas de renderização ou sobreposição de elementos. Além disso, há verificações para tolerar mudanças nas identificações das perguntas – o script consegue mapear as perguntas tanto pelo código (ex: "Q007") quanto pelo texto da pergunta, evitando dependência estrita do número, que pode variar [12] [18].

1. **Confirmação final:** Após preencher todas as abas, o script retorna para a tela de **Dados da Etapa** e realiza um único clique no botão **"Confirmar"** no rodapé [19]. Esse design (uma confirmação única no final) evita gravar parcialmente cada formulário separadamente; em vez disso, todas as alterações são gravadas juntas apenas se o processo inteiro completar com sucesso. Isso aumenta a atomicidade do processo – se algo der errado antes da confirmação final, o usuário pode corrigir manualmente sem ter vários saves incompletos no sistema.

2. **Logs e finalização:** O script implementa uma classe `Tee` para duplicar a saída em um arquivo de log, caso seja fornecida a opção `--log` [20] [21]. Também são exibidos tempos de execução de cada parte (mensagens `[TIMER]` impressas via um decorator de medição de tempo em funções críticas) para ajudar a diagnosticar desempenho [22] [23]. Ao término, o programa mantém o navegador aberto e solicita um **ENTER no console** para só então fechar – isso permite ao usuário inspecionar visualmente a etapa preenchida antes de encerrar a execução, útil para conferir se tudo foi inserido corretamente [24].

Em resumo, a arquitetura atual separa bem as responsabilidades: um módulo para infra (driver, navegação, login), um para lógica de negócio (plano de respostas), e módulos para interação detalhada com cada formulário. O fluxo é linear, com verificações importantes (login, tipo de PT) e uso intensivo de logs e impressões para acompanhamento.

Coerência da Lógica e Considerações Técnicas

A lógica do projeto mostra-se **coerente e robusta** diante do objetivo proposto. Alguns pontos que evidenciam isso:

- **Centralização das regras de negócio:** Todas as regras para decidir respostas e EPIs estão encapsuladas na geração do plano (módulo de plano), enquanto o módulo de preenchimento apenas segue esse "plano". Isso facilita a manutenção, pois se alguma regra de negócio mudar (por exemplo, qual EPI é obrigatório para determinado risco), basta atualizar no gerador de plano, sem precisar alterar a lógica de preenchimento em si.

- **Resiliência na interação web:** As funções de infra e preenchimento são escritas com cuidado para contornar problemas comuns de automação web. Por exemplo, ao procurar elementos, muitas vezes usam `WebDriverWait` com condições de presença e clique antes de interagir, evitando *race conditions*. Funções como `safe_find_element` e `wait_and_click` fazem

tentativas de localizar/clicar e capturam exceções de timeout, stale element etc., ao invés de assumirem que tudo deu certo de primeira. Para clicar em resultados de busca, o código percorre uma lista de possíveis XPaths (diferentes estruturas de resultado) até encontrar um elemento válido [25]. E se um elemento desaparece no momento do clique (exceção *StaleElementReference*), há novas tentativas implementadas com pequenas pausas [26]. Esses tratamentos aumentam a confiabilidade do fluxo frente a páginas dinâmicas ou lentidão da rede.

- **Execuções idempotentes:** O processo foi pensado para que, se executado mais de uma vez sobre a mesma PT, não cause efeitos colaterais extras. Como as respostas padronizadas serão as mesmas, a segunda execução tenderá a encontrar os formulários já preenchidos corretamente e não alterar nada (além de possivelmente agilizar por já estar tudo marcado). Por exemplo, a rotina de preenchimento da tabela de EPIs verifica se um item já está presente antes de adicioná-lo, evitando duplicações. Assim, o script pode ser rodado novamente sobre uma PT sem prejuízo, caso haja falha na primeira vez, por exemplo.

- **Riscos e dependências:** O ponto frágil, já mencionado, é a dependência de **identificadores fixos da interface** (textos e XPaths). Qualquer alteração na aplicação APLAT (como mudança de rótulo de um botão ou reordenação de campos) pode quebrar a automação. O autor já alerta no README que ajustes de seletores e regras serão necessários se o sistema mudar [3]. Para mitigar isso, uma sugestão do relatório é **externalizar esses seletores e regras para arquivos de configuração**, de modo que atualizá-los não exija modificar o código-fonte principal. Essa melhoria será considerada na reimplementação.

- **Segurança de credenciais:** O projeto suporta uso de *keyring* para não expor a senha no código ou em parâmetros de linha de comando, o que é positivo. O fluxo implementado (`attempt_auto_login`) busca a senha segura no keyring e só recorre ao login manual se não conseguir [27] [28]. Ainda assim, é importante que o novo projeto documente bem como configurar o keyring e evite guardar senhas em texto plano.

- **Logging e observabilidade:** Além de registros básicos via `print`, há o mecanismo de log em arquivo via `Tee` e marcação de tempo. Uma possível melhoria sugerida seria adotar a biblioteca **logging** do Python para um controle mais estruturado e níveis de log configuráveis (info, debug, warning etc.), ou até gerar logs em formato JSON para facilitar análises posteriores. Isso não estava implementado, mas é uma ideia para a nova versão.

Em suma, o código atual cumpre seu objetivo de forma robusta, porém **bastante específico** ao cenário atual (P-18) e um tanto extenso/complexo. Há espaço para refatoração e modularização melhor, que abordaremos ao recriar o projeto do zero. A seguir, será apresentado um **plano passo-a-passo para reconstruir essa automação**, com melhorias de organização e clareza, minimizando riscos de falhas futuras.

Plano Detalhado para Recriar o Projeto do Zero

Vamos agora elaborar um guia **didático e sequencial** para reconstruir a automação *APLAT – Trabalho a Quente* do zero. O objetivo é criar um novo projeto mais organizado, implementando cada parte gradualmente e testando ao longo do processo para garantir funcionalidade e evitar erros difíceis de depurar. Iremos incorporar sugestões de melhoria como separar configurações em arquivos próprios e outras boas práticas (exceto as que não forem prioridade agora, conforme indicado).

Premissa inicial: Você já criou e ativou um ambiente virtual (venv) no VS Code. Partiremos daí, garantindo as dependências, estruturando os arquivos, escrevendo o código módulo por módulo e realizando testes locais em cada etapa. O projeto final poderá ser versionado em um novo repositório GitHub conforme desejado.

1. Configurar o Ambiente e Dependências Básicas

Comece instalando os pacotes necessários no seu ambiente virtual. Abra um terminal com o venv ativado e execute:

```
pip install selenium~=4.11.0 keyring PyYAML
```

- **Selenium**: biblioteca de automação de browsers. A versão 4.x é necessária para suporte ao Edge Chromium [29].
- **Keyring**: para armazenamento seguro de senha (vamos utilizá-lo conforme no projeto original).
- **PyYAML**: para lidar com arquivos YAML (iremos usar um arquivo de configuração `.yaml` para regras, o que facilita ajustes futuros).

  Obs: Verifique se o pip instalou tudo corretamente e se não houve erros de compatibilidade.

Em seguida, **prepare o WebDriver do Edge** (msedgedriver). É fundamental que a versão do *driver* corresponda à versão do seu navegador Edge instalado. Conforme a documentação da Microsoft, as três primeiras partes do número de versão do driver devem ser iguais às do Edge [30] [31]. Para garantir isso: - Abra o Edge e navegue até `edge://settings/help` para ver o número da versão do navegador (por exemplo, 114.0.1823.67). - Acesse a página oficial de download do Edge WebDriver [32] e baixe o driver correspondente à sua versão (escolha o pacote correto, geralmente Windows x64 para desktops corporativos). - Extraia o executável **msedgedriver.exe** do zip baixado e coloque-o em um local de fácil acesso. Você tem duas opções: 1. **No PATH do sistema:** Coloque o executável em algum diretório já presente na variável de ambiente PATH, ou adicione o diretório dele ao PATH. Assim, o Selenium o encontrará automaticamente. 2. **No diretório do projeto:** Como o código original verifica alguns caminhos padrão, você pode colocar o *msedgedriver.exe* na raiz do seu projeto ou dentro da pasta do venv [33]. Por exemplo, pode criar uma pasta `drivers/` no projeto e colocar lá, ou mesmo na raiz ao lado dos arquivos .py. O código cuidará de procurar nesses lugares e usar o driver encontrado [33].

Certifique-se de que o driver funciona: uma forma simples de testar é executar no Python uma breve inicialização do Selenium. Por exemplo, no terminal Python do venv:

```
from selenium import webdriver
from selenium.webdriver.edge.service import Service as EdgeService
service = EdgeService('<CAMINHO_PARA_MSEDGEDRIVER.exe>')
driver = webdriver.Edge(service=service)
driver.get("https://www.google.com")
driver.quit()
```

Isso deve abrir rapidamente uma janela do Edge, navegar ao Google e fechar. Se der algum erro de versão incompatível, revise se o driver corresponde à versão do navegador.

Configurar credenciais no Keyring (opcional): Para permitir login automático sem digitar senha manualmente, armazene suas credenciais no *keyring* do sistema. Abra um console Python e execute:

```
import keyring
keyring.set_password("aplat.petrobras", "<seu_usuario>", "<sua_senha>")
```

Use o mesmo usuário que você usaria para login no APLAT. Isso salvará a senha no gerenciador de credenciais do seu SO, associada ao serviço "aplat.petrobras" (que é o nome padrão usado no código [34]). Mais tarde, ao rodar o script com `--use-keyring --user <seu_usuario>`, o código irá buscar essa senha do keyring automaticamente [27]. *Nota:* Caso prefira, pode pular este passo e fazer login manual quando o navegador abrir – o script dará essa opção se o keyring não estiver configurado.

2. Estruturar o Projeto e Arquivos Iniciais

Agora vamos criar a estrutura básica de arquivos e pastas do novo projeto. Uma boa organização (inspirada no projeto original, mas com melhorias) é a seguinte:

```
aplatquente/              # Diretório raiz do projeto (pode ser o repo Git)
├── config/
│   ├── xpaths.py         # Seletores XPaths e strings fixas da interface
APLAT
│   └── regras.yaml       # Regras e mapeamentos (ex: respostas padrão,
EPIs)
├── aplatquente.py        # Script principal (CLI e orquestração do
processo)
├── infra.py              # Módulo de infraestrutura (webdriver, login,
navegação)
├── plano.py              # Módulo de geração do plano de respostas (lógica
de negócio)
├── preenchimento.py      # Módulo de preenchimento automático dos
formulários
├── epi.py                # Módulo específico para tratamento da tabela de
EPIs
└── requirements.txt      # (Opcional) lista de dependências do projeto
```

Crie as pastas e arquivos acima (pode usar o explorador do VS Code ou comandos `mkdir` / `touch`). Em seguida, vamos populá-los passo a passo.

- `config/xpaths.py`: Abra este arquivo e defina nele todas as constantes de XPath e seletores utilizados na interface. Isso inclui botões, campos e elementos que o código precisa clicar ou ler. Você pode extrair esses valores do código original `quent1_infra.py` e outros, centralizando aqui. Por exemplo, adicione no `xpaths.py`:

```
# config/xpaths.py
# XPaths para elementos da interface APLAT (P-18) - Trabalho a Quente
# Botões principais
XPATH_BTN_EXIBIR_OPCOES = "//button[normalize-space()='Exibir opções']"
```

```
XPATH_BTN_PESQUISAR = "//button[normalize-space()='Pesquisar']"
XPATH_BTN_FECHAR = "//app-botoes-etapa//button[normalize-space()='Fechar']"
XPATH_BTN_CONFIRMAR = "//app-botoes-etapa//button[normalize-
space()='Confirmar']"
XPATH_BTN_OK = "//app-messagebox//button[normalize-space()='Ok']"
# botão Ok em diálogos modais

# Campos de formulário
XPATH_CAMPO_DATA = "//input[@placeholder='Selecione uma data']"
XPATH_CAMPO_NUMERO = "//input[@formcontrolname='numeroetapa']"
XPATH_PASSWORD_FIELD = "//input[@type='password' and not(@disabled)]"  #
campo de senha no login

# Abas de navegação (formulários)
# Observação: usaremos um template para localizar abas pelo texto, ex: função
goto_tab construirá o XPath
# Exemplo: f"//ul[contains(@class,'tabAplat')]//a[normalize-
space()='{tab_name}']"

# Indicadores para verificar tela principal carregada (pós-login)
MAIN_SCREEN_INDICATORS = [
    "//button[normalize-space()='Exibir opções']",
    "//a[normalize-space()='EPI']",
    "//h3[contains(.,'Cadastro de PT')]"
]

# Resultado de pesquisa - possíveis XPaths para o primeiro item
SEARCH_RESULT_XPATHS = [
    "(//app-grid//table/tbody/tr)[1]",
    "(//table//tbody//tr)[1]",
    "(//ul[contains(@class,'list-group')]//li[contains(@class,'listagem')])
[1]",
]
```

Estes são apenas alguns exemplos. Coloque todos os seletores relevantes que encontrar no código original. Por exemplo, XPaths para identificar perguntas do questionário PT pelo texto ou código, etc., podem ficar aqui também, se forem constantes. Centralizar os XPaths facilitará manutenção: se a UI mudar algum texto (“Pesquisar” virar “Buscar”, por exemplo), você ajusta só aqui.

- `config/regras.yaml`: Neste arquivo YAML, vamos guardar as **bases padrão de respostas** e algumas regras de contexto de forma externa ao código. Por exemplo, podemos definir:

```
# config/regras.yaml
epi_radios_base:
  # Identificador da pergunta EPI: Resposta padrão
  Q001_CINTO: "Não"
  Q002_VENT: "Não"
  Q003_COLETE: "Não"
  Q004_ILUM: "Não"
  Q005_DPA: "Sim"
```

```
  Q006_PROT_FACIAL: "Sim"

epis_categoria_base:
  Luvas:
    - "LUVA DE PROTEÇÃO CONTRA IMPACTOS MODELO II (3, 4, 3, 3, 'C',
'P')"
  Proteção Respiratória:
    - "NÃO APLICÁVEL"
  Vestimentas:
    - "DUPLA PROTEÇÃO AUDITIVA"
    - "EPI´s OBRIGATÓRIOS (CAPACETE, BOTA, PROT. AURIC. E UNIFORME)"
  Óculos:
    - "ÓCULOS AMPLA VISÃO"
    - "PROTETOR FACIAL"

qpt_base:
  Q001_MUDANCA: "Não"
  Q001_PERMANENCIA: "Não"
  Q002_ACOMP: "Sim"
  Q002_MANOBRAS: "NA"
  Q003_DRENADO: "NA"
  Q004_SINALIZADO: "NA"
  Q005_INSPECOES: "NA"
  Q006_COMB_INC: "NA"
  Q007_MANG_AR: "NA"
  Q008_LOCAL_ISOLADO: "Sim"
  Q009_FAGULHAS: "NA"
  Q010_ACOPLADO: "NA"
  Q011_TAMPONAMENTOS: "NA"
  Q012_RISCO_PP: "Não"
  Q013_INIBIR_SENSORES: "NA"
  Q014_OBSERVADOR: "NA"
```

Explicação: Estas seções correspondem às estruturas presentes no código original (`EPI_RADIOS_BASE`, `EPIS_CAT_BASE`, `QPT_BASE` etc. [35] [12]). Estamos nomeando as chaves de forma descritiva (por exemplo, "Q001_MUDANCA") para identificar cada pergunta do questionário PT ou EPI adicional. Os valores são as respostas padrão previstas.

Mais adiante, o código Python lerá este YAML e carregará esses dicionários para uso interno. Ter isso em um arquivo facilita ajustes sem alterar o código (por exemplo, se em outra plataforma a resposta padrão de alguma pergunta mudar de "NA" para "Não", basta editar o YAML).

> **Nota:** A identação em YAML é importante. Certifique-se de usar espaços corretamente conforme no exemplo. Além disso, alguns caracteres especiais em strings (como a presença de apóstrofo em **EPI´s** OBRIGATÓRIOS) foram incluídos como está, mas fique atento a escapar adequadamente caso necessário (neste caso, o YAML suporta apóstrofo

duplicado para escapar, mas aqui está dentro de aspas duplas então não conflita). Em caso de dúvidas, teste a leitura do YAML depois.

- Arquivos de código (`infra.py`, `plano.py`, `preenchimento.py`, `epi.py`): Comece criando esses arquivos vazios por ora. Vamos implementá-los passo a passo nas próximas etapas.

- `aplatquente.py`: Crie também o script principal vazio. Este será o ponto de entrada do programa, contendo o parse dos argumentos de linha de comando e a chamada sequencial das funções dos módulos.

A estrutura básica está pronta. A seguir, vamos implementar cada componente.

3. Implementar o Módulo de Infraestrutura (infra.py) – WebDriver, Login e Navegação Básica

No arquivo `infra.py`, vamos construir as funções responsáveis por configurar o WebDriver do Edge, realizar login (automático ou manual) e navegar entre telas. Muitas dessas funções corresponderão às do antigo `quent1_infra.py`, porém podemos simplificar ou melhorar onde necessário.

Abra `infra.py` e comece implementando o seguinte:

- **Configuração do WebDriver (Edge):**

```
# infra.py
from selenium import webdriver
from selenium.webdriver.edge.service import Service as EdgeService
from selenium.webdriver.edge.options import Options as EdgeOptions

import os, time
from typing import Optional
# (importações de utilidades do Selenium, WebDriverWait, By, EC, etc, serão
adicionadas mais abaixo conforme uso)

# Configurações de opções para o Edge (navegador)
EDGE_OPTIONS = [
    "--start-maximized",
    "--disable-gpu",
    "--disable-dev-shm-usage",
    "--no-sandbox"
]

def create_edge_driver() -> webdriver.Edge:
    """Cria e retorna uma instância do WebDriver do Edge com opções
configuradas."""
    options = EdgeOptions()
    options.add_experimental_option('excludeSwitches', ['enable-logging'])
    for arg in EDGE_OPTIONS:
        options.add_argument(arg)
```

```
    # Tenta encontrar o executável do msedgedriver em alguns locais comuns
    driver_paths = [
        ".venv/msedgedriver.exe",
        "msedgedriver.exe",
        os.path.join(os.getcwd(), "msedgedriver.exe")
    ]
    driver = None
    for path in driver_paths:
        if os.path.exists(path):
            try:
                print(f"[INFO] Usando msedgedriver encontrado em: {path}")
                service = EdgeService(path)
                driver = webdriver.Edge(service=service, options=options)
                break
            except Exception as e:
                print(f"[WARN] Falha ao iniciar WebDriver com {path}: {e}")
                driver = None
    if driver is None:
        # Tenta usar o driver no PATH do sistema
        try:
            driver = webdriver.Edge(options=options)
        except Exception as e:
            raise RuntimeError(f"Não foi possível iniciar o WebDriver Edge:
{e}")
    try:
        driver.maximize_window()
    except Exception:
        pass  # Se não conseguir maximizar (ex: em execução headless),
ignora.
    return driver
```

Essa função configura as opções do navegador (desabilitando logs, executando maximizado, etc.) e procura pelo executável do driver. Adicionamos alguns prints informativos. Basicamente é o que já existia [33] [36], mas certifique-se que *os caminhos correspondem a onde você colocou o msedgedriver*. Ajuste `driver_paths` se necessário (por ex., se optou pela pasta `drivers/`, inclua `"drivers/msedgedriver.exe"` na lista).

- **Funções utilitárias de espera e clique:** Após a importação de Selenium, adicione:

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException,
StaleElementReferenceException, ElementClickInterceptedException
from selenium.webdriver.common.action_chains import ActionChains

def wait_for_document_ready(driver, timeout: float):
    """Aguarda o carregamento completo do documento (estado
'complete')."""
    WebDriverWait(driver, timeout).until(
```

```
        lambda d: d.execute_script("return document.readyState") ==
"complete"
    )

def safe_find_element(driver, xpath: str, timeout: float):
    """Tenta localizar um elemento pelo XPath dentro do timeout, sem
lançar exceção em caso de falha."""
    try:
        return WebDriverWait(driver,
timeout).until(EC.presence_of_element_located((By.XPATH, xpath)))
    except TimeoutException:
        return None

def wait_and_click(driver, xpath: str, timeout: float, description: str
= ""):
    """Espera um elemento ficar clicável e realiza o clique."""
    desc = description or xpath
    elem = WebDriverWait(driver,
timeout).until(EC.element_to_be_clickable((By.XPATH, xpath)))
    try:
        elem.click()
    except Exception as e:
        # Se clique normal falhar, tenta JavaScript click
        driver.execute_script("arguments[0].click();", elem)
    print(f"[INFO] Clicou em {desc}.")
```

Aqui definimos: `wait_for_document_ready` (espera a página carregar totalmente), `safe_find_element` (retorna elemento ou None, evitando exceção se não achar), e `wait_and_click` (garante que o elemento está clicável antes de clicar, com fallback de JS). Esses padrões estavam no projeto original e ajudam a evitar problemas de timing.

- **Login Automático e Manual:** Precisamos implementar a lógica de login similar à função `attempt_auto_login` do original [37], juntamente com sub-funções para verificação de login e realização do mesmo. Faremos:

```
import getpass  # para solicitar senha do usuário caso necessário

def is_login_page_loaded(driver) -> bool:
    """Verifica se a página atual está pedindo login (campo de senha
visível)."""
    try:
        WebDriverWait(driver,
3).until(EC.visibility_of_element_located((By.XPATH, "//
input[@type='password' and not(@disabled)]")))
        return True  # campo de senha presente -> está na tela de login
    except TimeoutException:
        return False

def attempt_auto_login(driver, url: str, timeout: float, use_keyring:
bool = False, user: Optional[str] = None, keyring_service: str =
```

```
"aplat.petrobras") -> bool:
    """Tenta abrir a URL do APLAT e realizar login automático (via
keyring se possível). Retorna True se logado."""
    print(f"[INFO] Acessando APLAT: {url}")
    driver.get(url)
    wait_for_document_ready(driver, timeout)
    # Se já estiver autenticado (SSO ou sessão previa), a página
principal aparece sem login
    if not is_login_page_loaded(driver):
        print("[LOGIN] SSO ativo ou sessão prévia detectada – já
logado.")
        return True
    # Se precisamos logar:
    if use_keyring and user:
        try:
            import keyring
            pwd = keyring.get_password(keyring_service, user)
        except Exception as e:
            pwd = None
            print(f"[WARN] Keyring não disponível ou erro ao obter
senha: {e}")
        if pwd:
            print(f"[LOGIN] Tentando login automático com keyring para
usuário {user}...")
            if _perform_login(driver, user, pwd, timeout):
                return True
        else:
            print(f"[WARN] Senha não encontrada no keyring para
{user}.")
    # Se chegou aqui, login automático não foi bem-sucedido
    return False

def _perform_login(driver, username: str, password: str, timeout: float)
-> bool:
    """Preenche os campos de login e submete. Retorna True se conseguiu
entrar na tela principal."""
    try:
        # Preenche usuário e senha - assumindo campos padrão de SSO
Petrobras
        user_field = WebDriverWait(driver,
5).until(EC.element_to_be_clickable((By.ID, "username")))
        pass_field = WebDriverWait(driver,
5).until(EC.element_to_be_clickable((By.ID, "password")))
        user_field.clear(); user_field.send_keys(username)
        pass_field.clear(); pass_field.send_keys(password)
        # Tenta localizar e acionar o botão de login
        submit_buttons_xpaths = [
            "//button[normalize-space()='Entrar']",
            "//button[normalize-space()='Acessar']",
            "//input[@type='submit']",
            "//button[contains(.,'Sign in')]",
```

```
            "//button[contains(.,'Login')]"
        ]
        for xpath in submit_buttons_xpaths:
            btn = driver.find_elements(By.XPATH, xpath)
            if btn:
                try:
                    btn[0].click()
                    break
                except Exception:
                    driver.execute_script("arguments[0].click();",
btn[0])
                    break
        # Aguarda um indicador da tela principal após login
        if WebDriverWait(driver, timeout).until(lambda d:
any(d.find_elements(By.XPATH, xp) for xp in MAIN_SCREEN_INDICATORS)):
            print("[INFO] Login realizado com sucesso.")
            return True
    except Exception as e:
        print(f"[ERROR] Falha no processo de login automático: {e}")
    return False

def prompt_manual_login(driver, timeout: float):
    """Orienta o usuário a fazer login manualmente caso automático não
funcione."""
    print("[INFO] Por favor, realize o login manualmente no navegador
que abriu (SSO ou usuário/senha).")
    # Espera o usuário confirmar que logou
    try:
        input("Após realizar o login e visualizar a tela principal do
APLAT, pressione ENTER aqui...")
    except Exception:
        pass
    # Verifica se tela principal apareceu
    main_loaded = False
    try:
        WebDriverWait(driver, timeout).until(lambda d:
any(d.find_elements(By.XPATH, xp) for xp in MAIN_SCREEN_INDICATORS))
        main_loaded = True
    except TimeoutException:
        main_loaded = False
    if main_loaded:
        print("[INFO] Login manual confirmado, prosseguindo.")
    else:
        raise RuntimeError("Login manual não confirmado dentro do tempo
esperado.")
```

Vamos explicar: a função `attempt_auto_login` tenta usar o keyring se disponível; se não der certo, simplesmente retorna False (indicando que precisará de login manual). A `_perform_login` preenche o formulário; adaptamos os seletores para campos de login (usando IDs "username" e "password" típicos do SSO Petrobras – ajuste se necessário caso o APLAT tenha uma tela diferente). Também definimos alguns possíveis textos de botão de login

(Entrar, Acessar, etc.) e tentamos clicar [38] [39]. Depois esperamos por um dos elementos da tela principal (definidos em `MAIN_SCREEN_INDICATORS` lá no `xpaths.py`) para assegurar que o login teve sucesso.

A `prompt_manual_login` serve para quando o automático falhar ou não for usado – ela imprime instruções para o usuário fazer login manualmente no navegador aberto e esperar até que a tela principal carregue (ou um timeout ocorra). Se o usuário não conseguir logar, o script aborta com erro.

> **Importante:** Verifique se os seletores de usuário/senha e botões correspondem exatamente ao sistema de login do APLAT. Se estiver usando SSO corporativo web, pode ser que não haja campos "username" e "password" diretamente – talvez o login já ocorra via Windows logon ou certificado. Nesse caso, a detecção `is_login_page_loaded` pode indicar False (já logado) imediatamente, e seguir em frente. Ajuste esses detalhes conforme o comportamento real observado.

- **Funções de Pesquisa e Navegação por Abas:** Ainda em `infra.py`, implementamos agora a parte de pesquisar a etapa e abrir as abas adequadas.

```
def perform_search(driver, data_str: str, numero_etapa: str, timeout:
float, search_timeout: float, detail_wait: float):
    """Executa a busca da etapa no APLAT e abre o primeiro resultado
encontrado."""
    print(f"[INFO] Pesquisando etapa {numero_etapa} na data
{data_str}...")
    # Pressiona "Exibir opções" para mostrar filtros de pesquisa
    wait_and_click(driver, XPATH_BTN_EXIBIR_OPCOES, timeout, "botão
Exibir Opções")
    # Preenche campo de data
    date_field = WebDriverWait(driver,
timeout).until(EC.element_to_be_clickable((By.XPATH, XPATH_CAMPO_DATA)))
    date_field.clear(); date_field.send_keys(data_str)
    print(f"[INFO] Data preenchida: {data_str}")
    # Preenche número da etapa
    num_field = WebDriverWait(driver,
timeout).until(EC.element_to_be_clickable((By.XPATH,
XPATH_CAMPO_NUMERO)))
    num_field.clear(); num_field.send_keys(numero_etapa)
    print(f"[INFO] Número da etapa preenchido: {numero_etapa}")
    # Clica em pesquisar
    wait_and_click(driver, XPATH_BTN_PESQUISAR, timeout, "botão
Pesquisar")
    # Aguarda resultados
    try:
        result = WebDriverWait(driver, search_timeout).until(lambda d:
_find_first_result(d))
    except TimeoutException:
        raise RuntimeError(f"Nenhum resultado encontrado para etapa
{numero_etapa} na data {data_str}.")
    # Clica no primeiro resultado encontrado
    row_element, xpath_used = result
```

```
    try:
        row_element.click()
    except Exception:
        driver.execute_script("arguments[0].click();", row_element)
    print(f"[INFO] Resultado da etapa {numero_etapa} aberto (XPath
usado: {xpath_used}).")
    time.sleep(detail_wait)  # pequena espera para garantir que os
detalhes carregaram

def _find_first_result(driver):

"""Busca pelo primeiro elemento de resultado de pesquisa disponível,
usando os XPaths conhecidos."""
    for xpath in SEARCH_RESULT_XPATHS:
        elements = driver.find_elements(By.XPATH, xpath)
        for elem in elements:
            if elem.is_displayed() and elem.is_enabled():
                return (elem, xpath)
    return False  # sinal para WebDriverWait continuar esperando

def goto_tab(driver, tab_name: str, timeout: float):
    """Navega para uma aba específica identificada pelo texto do
rótulo."""
    xpath = f"//ul[contains(@class,'tabAplat')]//a[normalize-
space()='{tab_name}']"
    # Se a aba já estiver ativa, nada a fazer
    active = driver.find_elements(By.XPATH, xpath + "/
parent::li[@class='active']")
    if active:
        return
    # Caso contrário, tenta clicar na aba
    tab_link = WebDriverWait(driver,
timeout).until(EC.element_to_be_clickable((By.XPATH, xpath)))
    try:
        tab_link.click()
    except Exception:
        driver.execute_script("arguments[0].click();", tab_link)
    # Opcional: aguardar um elemento específico da aba carregar, se
houver.
    time.sleep(0.5)
```

A função `perform_search` segue de perto o original: clica em "Exibir opções", preenche data e número [40], clica em Pesquisar [41], depois aguarda e clica no primeiro resultado [42] [43]. Utilizamos uma função auxiliar `_find_first_result` para testar múltiplos XPaths de resultado (caso a tela de listagem mude). A `goto_tab` simplesmente constrói o XPath da aba pelo nome (por exemplo "Questionário PT", "EPI", etc.) e clica, a menos que já esteja na aba ativa. Esse método de navegação por texto de aba é flexível e foi usado no original [44].

- **Funções de Fechar/Confirmar Etapa:** Também do infra, podemos implementar funções para fechar modais ou confirmar a PT:

```
def fechar_modal_etapa(driver, timeout: float):
    """Fecha o modal de etapa (após confirmar) se aberto."""
    try:
        WebDriverWait(driver,
timeout).until(EC.element_to_be_clickable((By.XPATH,
XPATH_BTN_FECHAR))).click()
        print("[INFO] Modal de etapa fechado.")
    except TimeoutException:
        pass

def clicar_botao_confirmar_rodape(driver, timeout: float):
    """Clica no botão 'Confirmar' no rodapé da etapa."""
    btn = WebDriverWait(driver,
timeout).until(EC.element_to_be_clickable((By.XPATH,
XPATH_BTN_CONFIRMAR)))
    try:
        btn.click()
    except Exception:
        driver.execute_script("arguments[0].click();", btn)
    # Aguarda eventualmente um popup/modal de confirmação extra e clica
OK se aparecer
    try:
        WebDriverWait(driver,
5).until(EC.element_to_be_clickable((By.XPATH, XPATH_BTN_OK))).click()
    except TimeoutException:
        pass
    print("[INFO] Botão Confirmar acionado e confirmado (se
necessário).")
```

Essas rotinas cuidam de clicar no botão **Confirmar** e tratar um possível diálogo "OK" de confirmação, e de fechar a janela da etapa (caso o sistema abra uma modal de sucesso ou mantenha a aba aberta). No original havia algo semelhante (funções `fechar_modal_etapa` e `clicar_botao_confirmar_rodape`).

Com isso, o módulo `infra.py` está implementado com as funcionalidades principais: iniciar o browser, fazer login, pesquisar etapas, navegar por abas e confirmar.

Teste incremental – Login e Pesquisa: Antes de prosseguir, é recomendável testar se essa infraestrutura funciona até a abertura de uma etapa. Podemos criar temporariamente no final de `infra.py` um pequeno trecho para rodar um teste quando o arquivo for executado diretamente (usando `if __name__ == "__main__":`). Porém, como temos o script principal para isso, podemos já utilizar o `aplatquente.py` para testar.

Por enquanto, crie no `aplatquente.py` um esqueleto mínimo:

```
# aplatquente.py
import sys, argparse
from infra import create_edge_driver, attempt_auto_login,
prompt_manual_login, perform_search
```

```
def parse_args():
    parser = argparse.ArgumentParser(description="Automação APLAT - Trabalho a Quente")
    parser.add_argument("--valor", "-v", nargs="+", help="Número(s) da etapa a processar", required=True)
    parser.add_argument("--data", "-d", help="Data da programação (formato YYYY-MM-DD)", required=True)
    parser.add_argument("--use-keyring", action="store_true", help="Usar keyring para senha (requer --user)")
    parser.add_argument("--user", help="Usuário para login (obrigatório se usar --use-keyring)")
    parser.add_argument("--keyring-service", default="aplat.petrobras", help="Nome do serviço no keyring (default: aplat.petrobras)")
    parser.add_argument("--timeout", type=float, default=30.0, help="Timeout em segundos para operações padrão")
    parser.add_argument("--search-timeout", type=float, default=30.0, help="Timeout para busca de etapa")
    parser.add_argument("--url", default="https://aplat.petrobras.com.br/#/permissaotrabalho/P-18/planejamento/programacaodiaria", help="URL do APLAT")
    return parser.parse_args()

if __name__ == "__main__":
    args = parse_args()
    driver = create_edge_driver()
    # Tentativa de login automático
    logged_in = attempt_auto_login(driver, args.url, args.timeout, use_keyring=args.use_keyring, user=args.user, keyring_service=args.keyring_service)
    if not logged_in:
        try:
            prompt_manual_login(driver, args.timeout)
        except Exception as e:
            print(f"[ERROR] Login falhou: {e}")
            driver.quit()
            sys.exit(1)
    # Realiza a pesquisa e abertura de cada etapa
    for etapa in args.valor:
        try:
            perform_search(driver, args.data, etapa, args.timeout, args.search_timeout, detail_wait=0.3)
            print(f"[INFO] Etapa {etapa} aberta com sucesso.")
        except Exception as e:
            print(f"[ERROR] Falha ao buscar/abrir etapa {etapa}: {e}")
    # (iremos completar o restante do fluxo nas próximas etapas)
    input("Pressione ENTER para encerrar...")
    driver.quit()
```

No código acima, estamos parseando argumentos (similar ao original) e testando: - Abre o driver - Faz login (tenta auto, senão manual) - Percorre cada valor de etapa fornecido (`--valor` pode ser um ou vários números) e tenta abrir a etapa.

Rodando esse script com, por exemplo:

```
python aplatquente.py --valor 12345 --data 2023-10-01 --use-keyring --user
SEUUSUARIO
```

(deve substituir por uma etapa real existente no sistema e data correspondente), esperamos que: o Edge abra, faça login (automático se configurou keyring ou manual se não), e depois pesquise pela etapa 12345 do dia 01/10/2023, abrindo-a. Se der tudo certo, verá no console mensagens de sucesso e o navegador deve exibir a PT aberta na tela de **Dados da Etapa**.

Verifique se esse processo funciona antes de prosseguir. Caso encontre problemas: - Se o login automático não funcionar, tente sem `--use-keyring` e faça manual, só para isolar se o problema é no keyring ou na lógica de detecção de tela. - Se não encontrar resultados de pesquisa, confira se a data e número estão corretos e no formato esperado. Lembre que a data deve estar no padrão que o campo aceita (talvez DD/MM/YYYY ou YYYY-MM-DD dependendo da localidade; adaptamos para AAAA-MM-DD que geralmente funciona). - Se mesmo campos corretos nada acontece, pode ser que o XPath do botão ou campo esteja errado – inspecione no DevTools do navegador e ajuste os seletores no `config/xpaths.py` se necessário.

Até aqui, **temos a parte de infra e navegação funcional**. Uma vez que conseguir abrir uma etapa com sucesso, feche o navegador e vamos para a próxima parte.

4. Implementar Geração do Plano de Trabalho (plano.py)

Agora vamos criar o módulo `plano.py`, que será responsável por ler as informações da etapa e determinar as respostas que precisamos preencher. Esse módulo corresponde ao coração da lógica de negócio (similar ao `quent2_plano.py` antigo).

As principais responsabilidades aqui são: - **Coletar dados da página**: descrição da etapa e características do trabalho. - **Normalizar textos**: remover acentos, colocar em maiúsculas, etc., para facilitar busca por palavras-chave. - **Montar contexto de flags**: a partir dos textos, definir variáveis booleanas `tem_*` indicando a presença de certos riscos/condições (chama aberta, altura, espaço confinado, pressurizado, etc.). - **Gerar bases de respostas**: carregar do arquivo `regras.yaml` as bases padrão (dicionários) e ajustá-las conforme o contexto (por exemplo, se `tem_altura=True`, alterar algumas respostas de "Não" para "Sim"). - **Consolidar tudo em uma estrutura (por exemplo, um dict) que será usada nas funções de preenchimento.**

Vamos implementar isso passo a passo:

Abra `plano.py` e escreva:

- **Imports e leitura das regras**:

```
import re, unicodedata, yaml
from typing import Dict, Any, Tuple, List
from config import xpaths  # podemos precisar de XPaths para coleta se
```

```
formos usar aqui
from infra import safe_find_element
```

Usaremos o `yaml` (PyYAML) para carregar o arquivo de regras. Então logo após, carregue o arquivo YAML:

```
# Carrega as bases de regras do YAML
try:
    with open("config/regras.yaml", "r", encoding="utf-8") as f:
        REGRAS = yaml.safe_load(f)
except Exception as e:
    raise RuntimeError(f"Erro ao carregar config/regras.yaml: {e}")
# Extrai as seções do YAML para variáveis Python
EPI_RADIOS_BASE: Dict[str, str] = REGRAS.get("epi_radios_base", {})
EPIS_CATEGORIA_BASE: Dict[str, List[str]] =
REGRAS.get("epis_categoria_base", {})
QPT_BASE: Dict[str, str] = REGRAS.get("qpt_base", {})
```

Agora, `EPI_RADIOS_BASE` será um dicionário onde as chaves são strings como "Q001_CINTO" e valores "Sim/Não". O mesmo para `QPT_BASE`. As EPIs por categoria `EPIS_CATEGORIA_BASE` é um dict de listas (vamos converter listas para sets depois para facilitar operações de conjunto).

- **Função de normalização de texto** (retirar acentos, deixar caixa alta, etc.):

```
def normalizar_texto(texto: str) -> str:
    """Normaliza o texto removendo acentos, colocando em maiúsculas e
trim dos espaços."""
    if not texto:
        return ""
    # Remove acentos
    nfkd = unicodedata.normalize("NFD", texto)
    texto_sem_acento = "".join(ch for ch in nfkd if not
unicodedata.combining(ch))
    # Caixa alta
    texto_upper = texto_sem_acento.upper()
    # Substitui qualquer sequência de espaços/brancos por um único
espaço
    texto_limpo = re.sub(r"\s+", " ", texto_upper).strip()
    return texto_limpo
```

Similar à original [45].

- **Coleta de descrição da etapa**: Precisamos encontrar o campo de descrição. No original, usam diversos XPaths candidatos [46]. Podemos replicar isso:

```
def coletar_descricao(driver, timeout: float) -> str:
    """Tenta obter o texto da Descrição da Etapa."""
```

```
    candidatos = [
        "//app-dados-da-etapa//
textarea[contains(@formcontrolname,'descricao')]",
        "//textarea[@id='descricao']",
        "//textarea[contains(@formcontrolname,'descricao')]"
    ]
    for xp in candidatos:
        el = safe_find_element(driver, xp, timeout)
        if el:
            texto = (el.get_attribute("value") or el.text or "").strip()
            if texto:
                print(f"[DEBUG] Descrição encontrada via XPath: {xp}")
                return texto
    print("[WARN] Não foi possível localizar a Descrição da Etapa.")
    return ""
```

Essa função tenta 3 variações de XPaths comuns para o campo de texto de descrição. Retorna a string da descrição ou `""` se não achar nada, emitindo um warning.

- **Coleta de características do trabalho**: Esta é mais complexa, pois a interface pode listar as características de formas diferentes. No original, temos 3 métodos:

- Método 1: vários `<span>` com classe `nomecaracteristica` dentro de um componente.
- Método 2: um `<fieldset>` cujo texto interno contém todas características.
- Método 3: usar regex no texto completo da página procurando pelo trecho após "Características do trabalho -".

Podemos implementar da mesma forma:

```
def coletar_caracteristicas(driver, timeout: float) -> str:
    """Retorna as características do trabalho (concatenadas por vírgula) da
etapa atual."""
    car_list: List[str] = []
    # Método 1: busca spans listando características selecionadas
    try:
        spans = driver.find_elements(By.XPATH, "//app-input-caracteristicas//
span[@class='nomecaracteristica']")
        for span in spans:
            texto = span.text.strip()
            if texto and texto not in car_list:
                car_list.append(texto)
        if car_list:
            resultado = ", ".join(car_list)
            print(f"[DEBUG] Características encontradas (método 1):
{resultado}")
            return resultado
    except Exception as e:
        print(f"[DEBUG] Método 1 falhou: {e}")
    # Método 2: procura um fieldset contendo "Características do trabalho"
    try:
```

```
        fieldset = safe_find_element(driver, "//fieldset[contains(.,
'Características do trabalho')]", timeout)
        if fieldset:
            texto = fieldset.text
            linhas = [lin.strip() for lin in texto.splitlines() if
lin.strip()]
            for lin in linhas:
                if "Características do trabalho" in lin:
                    continue
                if lin and lin not in car_list:
                    car_list.append(lin)
            if car_list:
                resultado = ", ".join(car_list)
                print(f"[DEBUG] Características encontradas (método 2):
{resultado}")
                return resultado
    except Exception as e:
        print(f"[DEBUG] Método 2 falhou: {e}")
    # Método 3: regex no texto completo da aba Dados da Etapa
    try:
        container = safe_find_element(driver, "//app-dados-da-etapa",
timeout)
        if container:
            full_text = container.text
            match = re.search(r"Características do trabalho\s*-\s*(.*?)(?
=\n\s*\n|\n[A-ZÀ-Ú]|$)", full_text, re.DOTALL | re.IGNORECASE)
            if match:
                bloco = match.group(1).strip()
                # Remove alguns símbolos especiais (se houver setas ou
marcadores)
                bloco = re.sub(r"[\u25b6\u25c0\u25b2\u25bc]", "",
bloco).strip()
                linhas = [ln.strip() for ln in bloco.splitlines() if
ln.strip()]
                if linhas:
                    resultado = ", ".join(linhas)
                    print(f"[DEBUG] Características encontradas (método 3):
{resultado}")
                    return resultado
    except Exception as e:
        print(f"[WARN] Método 3 falhou ao extrair características: {e}")
    # Se nada encontrado:
    print("[WARN] Nenhuma característica do trabalho encontrada.")
    return ""
```

Importante: usamos `driver.find_elements` e `By` dentro desta função, então precisamos importar `By` no topo (já foi importado em infra e podemos importar de lá ou do selenium direto aqui também). Para simplicidade, adicione `from selenium.webdriver.common.by import By` no topo de `plano.py` também.

- **Montar contexto de flags tem_***: A função a seguir analisará a descrição e características (já normalizadas) para definir flags booleans. Baseado no original [47] [48] e seguintes, podemos fazer:

```
def montar_contexto(descricao: str, caracteristicas: str) -> Dict[str, bool]:
    """Define flags de contexto (tem_alguma_coisa=True/False) com base na descrição e características fornecidas."""
    texto_full = f"{normalizar_texto(descricao)} {normalizar_texto(caracteristicas)}"
    ctx = {}
    ctx["texto_full"] = texto_full  # talvez útil manter o texto consolidado
    # Flags de exemplo (podemos expandir conforme necessário):
    ctx["tem_espaco_confinado"] = bool(re.search(r"\b(ESPA[CÇ]O CONFINADO|INTERIOR DE ESPA[CÇ]O|DENTRO DE|INTERIOR DO)\b", texto_full))
    ctx["tem_altura"] = bool(re.search(r"\b(TRABALHO EM ALTURA|ACESSO POR CORDAS|CORDAS|NR-35|ALTURA)\b", texto_full))
    ctx["tem_acesso_cordas"] = "ACESSO POR CORDAS" in texto_full
    ctx["tem_sobre_o_mar"] = "SOBRE O MAR" in texto_full
    ctx["tem_chama"] = bool(re.search(r"\b(CHAMA ABERTA|OXICORTE|SOLDA|ESMERILHADEIRA)\b", texto_full))
    ctx["tem_trat_mec"] = "TRATAMENTO MECÂNICO" in texto_full or "TRAT. MEC" in texto_full
    # (Acrescente aqui outras flags relevantes: tem_hidrojato, tem_partes_moveis, tem_pressurizado, etc.,
    # usando padrões baseados nas palavras-chave que aparecem na descrição/características)
    # Exemplo:
    ctx["tem_hidrojato"] = "JATO DE ÁGUA" in texto_full or "HIDROJATEAMENTO" in texto_full
    ctx["tem_partes_moveis"] = "PARTES MÓVEIS" in texto_full or "PARTES MOVEIS" in texto_full
    # ... (continue com outras condições conforme necessidade)
    return ctx
```

Observação: Aqui incluímos algumas detecções comuns. O original tinha várias flags definidas explicitamente. Você pode adicionar todas as que julgar necessárias baseadas no domínio do problema (por exemplo, tem_pressurizado, tem_eletricidade, etc., se aplicável). Use regex ou `in` apropriado para detectar termos no `texto_full`. Garantir que `texto_full` está normalizado (maiúsculo sem acento) ajuda a acertar essas buscas.

- **Montar bases específicas ajustadas ao contexto:** Usaremos as bases carregadas do YAML e alterar conforme as flags:

```
def ajustar_base_epi_radios(ctx: Dict[str, bool]) -> Dict[str, str]:
    base = EPI_RADIOS_BASE.copy()
    # Exemplo de ajustes:
```

```
    if ctx.get("tem_altura") or ctx.get("tem_acesso_cordas") or
ctx.get("tem_sobre_o_mar"):
        base["Q001_CINTO"] = "Sim"
    if ctx.get("tem_sobre_o_mar"):
        base["Q003_COLETE"] = "Sim"

# Ajuste de proteção facial: se houver riscos aos olhos (chama, trat
mecânico, etc)
    hazard_olhos = ctx.get("tem_chama") or ctx.get("tem_trat_mec") or
ctx.get("tem_lixadeira") or ctx.get("tem_corte")
    # ^ obs: você precisará definir tem_lixadeira, tem_corte etc. nas
flags se quiser usá-las.
    if hazard_olhos:
        base["Q006_PROT_FACIAL"] = "Sim"
    else:
        base["Q006_PROT_FACIAL"] = "Não"
    return base

def ajustar_base_epis_categoria(ctx: Dict[str, bool]) -> Dict[str, set]:
    # Convert list to set for easier addition/removal
    base = {cat: set(itens) for cat, itens in
EPIS_CATEGORIA_BASE.items()}
    hazard_olhos = ctx.get("tem_chama") or ctx.get("tem_trat_mec") or
ctx.get("tem_lixadeira") or ctx.get("tem_corte")
    if not hazard_olhos:

# Se não há risco a olhos, usar apenas óculos simples (substitui óculos
ampla visão/prot facial)
        base["Óculos"] = {"ÓCULOS SEGURANÇA CONTRA IMPACTO"}
    if ctx.get("tem_chama"):
        base.setdefault("Luvas", set()).update({"LUVA ARAMIDA",
"LUVA DE RASPA"})
    # (Acrescente outros ajustes conforme regras, ex: se tem_altura ->
adicionar cinto na lista de EPIs?
    # Porém EPI de cinto é mais um item de EPI Adicional, não da lista
de categorias.
    # A lista de categorias são EPIs padrão que devem estar na PT.)
    return base

def ajustar_base_qpt(ctx: Dict[str, bool]) -> Dict[str, str]:
    base = QPT_BASE.copy()
    # Exemplo: se "mudança" for caracterizado de alguma forma no texto,
alterar Q001_MUDANCA para "Sim".
    # (No original, parece que sempre "Não" para mudança, então não há
muito ajuste aqui.)
    # Acrescente ajustes caso alguma flag mude respostas do
questionário.
    return base

def ajustar_base_apn1(ctx: Dict[str, bool]) -> Dict[str, str]:
    # APN-1: perguntas Q001 a Q020 sim/não sobre tipos de risco.
```

```
    # Podemos derivar respostas baseando-nos no contexto.
    # Exemplo hipotético:
    base = {}
    base["Q001"] = "Sim" if ctx.get("tem_chama") else "Não"
    base["Q002"] = "Sim" if ctx.get("tem_gas") else "Não"
    # ... etc., conforme mapeamento de riscos APN-1 que você tenha.
    # Por ora, vamos supor que tudo que não detectamos é "Não".
    for i in range(1, 21):
        q = f"Q{str(i).zfill(3)}"
        if q not in base:
            base[q] = "Não"
    return base
```

Aqui, colocamos alguns exemplos de regras. Você deve complementar de acordo com as regras de negócio reais:

- A função `ajustar_base_epi_radios` usa as flags para mudar as respostas "Sim/Não" das perguntas de EPI adicional (Q001 cinto, Q003 colete, etc.) conforme as condições semelhantes ao original [13]. Preenchemos algumas: se tem trabalho em altura, cinto = Sim; se sobre o mar, colete = Sim; se riscos aos olhos, protetor facial = Sim, senão Não.
- `ajustar_base_epis_categoria` altera a lista de EPIs por categoria. Por exemplo, se não houver risco aos olhos, em vez de óculos ampla visão e protetor facial, usa óculos de segurança simples [49]. Se há chama, adiciona luvas específicas resistentes a calor.
- `ajustar_base_qpt` está meio vazia porque no original a base QPT já contemplava todas perguntas e aparentemente não mudava muito com contexto (exceto talvez acompanhamento periódico se tem espaço confinado, etc., mas não foi claro). Podemos deixar sem ajustes ou adicionar algum se souber de alguma regra (por ex: se tem espaço confinado, alguma pergunta de ventilação poderia virar "Sim"?).

- `ajustar_base_apn1` deveria mapear cada pergunta APN-1 (que são 20 perguntas sobre tipos de risco: inflamáveis, elétricos, espaço confinado, etc.) para Sim/Não baseado no contexto. Seria ideal definir claramente: por exemplo, se `tem_espaco_confinado=True`, então a pergunta "Trabalho em espaço confinado?" (digamos Q010) seria "Sim". Como não temos o mapeamento exato aqui, coloquei apenas um esquema e deixei padrão "Não" pra tudo não mapeado. Você deve ajustar conforme a realidade da APN-1 em trabalhos a quente.

- **Função principal de geração do plano:** finalmente, criamos uma função que amarra tudo e retorna as informações que precisamos:

```
def gerar_plano_trabalho_quente(driver, timeout: float) -> Dict[str,
Any]:
    """Coleta dados da etapa atual via driver e gera o plano de
respostas para trabalho a quente."""
    # 1. Coleta descrição e características da página
    descricao = coletar_descricao(driver, timeout)
    caracteristicas = coletar_caracteristicas(driver, timeout)
    # 2. Monta contexto de flags
    ctx = montar_contexto(descricao, caracteristicas)
    # 3. Ajusta bases conforme contexto
    plano = {}
```

```
    plano["ctx"] = ctx
    plano["descricao"] = descricao
    plano["caracteristicas"] = caracteristicas
    plano["qpt"] = ajustar_base_qpt(ctx)
    plano["epi_radios"] = ajustar_base_epi_radios(ctx)
    plano["epis_cat"] = ajustar_base_epis_categoria(ctx)
    plano["apn1"] = ajustar_base_apn1(ctx)
    return plano

def imprimir_plano(plano: Dict[str, Any]):
    """Imprime de forma organizada o plano gerado (contexto e respostas
esperadas)."""
    print("====== PLANO DE TRABALHO A QUENTE GERADO ======")
    # Contexto:
    print("Contexto de flags:")
    for k, v in plano.get("ctx", {}).items():
        if k != "texto_full":
            print(f"  - {k}: {v}")
    # Questionário PT:
    print("\nRespostas Questionário PT:")
    for q, resp in plano.get("qpt", {}).items():
        print(f"  - {q}: {resp}")
    # EPI adicional:
    print("\nEPI Adicional necessário (radios):")
    for epi, resp in plano.get("epi_radios", {}).items():
        print(f"  - {epi}: {resp}")
    # APN-1:
    print("\nRespostas APN-1:")
    for q, resp in plano.get("apn1", {}).items():
        print(f"  - {q}: {resp}")
    # EPIs por categoria:
    print("\nEPIs por categoria a selecionar:")
    for categoria, itens in plano.get("epis_cat", {}).items():
        print(f"  - {categoria}: {', '.join(itens) if isinstance(itens,
(list, set)) else itens}")
    print("================================================")
```

Com o módulo `plano.py` implementado, vamos integrá-lo ao fluxo principal e testar a geração do plano em uma etapa aberta.

Integração no script principal para teste: Volte ao `aplatquente.py` e importe as funções necessárias:

```
from plano import gerar_plano_trabalho_quente, imprimir_plano
```

No loop onde processamos as etapas, após `perform_search` abrir a etapa, chame:

```
plano = gerar_plano_trabalho_quente(driver, args.timeout)
imprimir_plano(plano)
```

Isso deve gerar o plano para aquela etapa e imprimir no console. Podemos por enquanto **não preencher nada**, só visualizar o plano, para verificar se as regras estão produzindo o esperado.

Então o bloco dentro do `for etapa in args.valor` fica assim:

```
for etapa in args.valor:
    try:
        perform_search(driver, args.data, etapa, args.timeout,
args.search_timeout, detail_wait=0.3)
        print(f"[INFO] Etapa {etapa} aberta com sucesso.")
    except Exception as e:
        print(f"[ERROR] Falha ao buscar/abrir etapa {etapa}: {e}")
        continue
    # Só prossegue se a etapa for Trabalho a Quente:
    # Podemos aqui validar o tipo de trabalho lendo o combo "Tipo Trabalho":
    tipo_valor = ""  # (Opcional: implementar coleta do tipo de trabalho
similar ao original coletar_tipo_trabalho)
    # Supondo que se abriu a etapa já é a correta, ou poderíamos implementar
uma checagem como no original:
    # if not is_trabalho_quente(tipo_valor): continue
    plano = gerar_plano_trabalho_quente(driver, args.timeout)
    imprimir_plano(plano)
    # (Preenchimento virá na próxima etapa)
```

> **Dica:** No trecho acima, idealmente implementaríamos uma função `is_trabalho_quente(driver)` para ler o campo "Tipo de Trabalho" na aba Dados da Etapa e verificar se contém "TRABALHO A QUENTE" [50]. Por simplicidade, se assumirmos que o usuário só fornece etapas de TQ, pode ignorar; mas pelo rigor, seria bom implementar. Poderia usar a função `safe_find_element` buscando o combo `tipoPT` e pegando a opção selecionada, similar ao original [51] [52].

Agora rode novamente o script para uma etapa conhecida de Trabalho a Quente (ex.: `--valor 12345 --data 2023-10-01`). Dessa vez, após abrir a etapa, o programa deverá imprimir um relatório do plano gerado. Confira se: - As flags de contexto (`tem_*`) fazem sentido conforme a descrição da etapa. - As respostas listadas para Questionário PT, EPI adicional, APN-1 e EPIs de categoria condizem com o que você esperaria para aquela etapa.

Se algo parecer incorreto, ajuste as regras no YAML ou nas funções de ajuste. Por exemplo, se deveria marcar alguma pergunta como "Sim" mas saiu "Não", identifique que flag ou condição deveria ter disparado. Use os prints `[DEBUG]` que colocamos para ver quais métodos de coleta foram usados e quais palavras-chave detectadas. Isso ajuda a calibrar a detecção de contexto.

Uma vez satisfeito com a geração do plano, seguimos adiante.

5. Implementar Rotinas de Preenchimento Automático (preenchimento.py e epi.py)

Esta etapa consiste em escrever as funções que efetivamente interagem com a página para marcar opções e inserir dados conforme o plano gerado. Vamos separar em dois módulos conforme a lógica original: - `preenchimento.py`: lida com **Questionário PT**, **EPI adicional**, **Análise Ambiental** e **APN-1** – basicamente formulários de perguntas com radios Sim/Não/NA. - `epi.py`: lida especificamente com a aba de **EPIs por Categoria**, que envolve marcar itens em listas de múltipla seleção (possivelmente checkboxes ou toggles).

Comecemos com `preenchimento.py`:

Abra o arquivo e adicione:

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import StaleElementReferenceException
import time
from infra import goto_tab, safe_find_element
```

- **Preenchimento do Questionário PT:** No sistema, o questionário PT é uma lista de perguntas numeradas (Q001, Q002, ...). O código original tinha uma classe para processar isso mapeando tanto por código quanto texto 12 18 . Podemos simplificar criando uma função que:
- Recebe o plano (dicionário de respostas `qpt`).
- Navega para a aba "Questionário PT".
- Itera pelas perguntas esperadas e marca a resposta correspondente.
- Faz alguma verificação se possível (ex.: checar se marcou corretamente ou se pergunta não encontrada).

Exemplo de implementação:

```
def preencher_questionario_pt(driver, plano_qpt: dict, timeout: float):
    """Marca as respostas no Questionário PT conforme o plano fornecido."""
    goto_tab(driver, "Questionário PT", timeout)
    for codigo_texto, resposta in plano_qpt.items():
        # codigo_texto pode ser algo como "Q001_MUDANCA" ou "Q002_ACOMP" do
nosso plano.
        # Vamos extrair apenas o código (ex: "Q001") para localizar no HTML.
        codigo = codigo_texto.split('_')[0] if '_' in codigo_texto else
codigo_texto
        # Tenta localizar a linha da pergunta pelo código:
        xpath_radio = f"//tr[td//text()[contains(.,'{codigo}')]]//
input[@type='radio' and @value='{resposta}']"
        elem = safe_find_element(driver, xpath_radio, 2)
        if elem:
            try:
                elem.click()
            except Exception:
                driver.execute_script("arguments[0].click();", elem)
```

```
        else:
            # Tenta localizar por texto da pergunta caso código não funcione
            # (Para isso precisaríamos ter o texto completo associado.
Supondo que QPT_BASE keys continham parte do texto.)
            # Exemplo: se codigo_texto="Q002_ACOMP", talvez o texto contido
seja "Acompanhamento Periódico".
            partes = codigo_texto.split('_', 1)
            if len(partes) > 1:
                trecho_texto = partes[1]  # assume a parte após o underscore
é uma pista do texto
                alt_xpath = f"//tr[td//text()
[contains(.,'{trecho_texto.split()[0]}')]]//input[@type='radio' and
@value='{resposta}']"
                elem = safe_find_element(driver, alt_xpath, 2)
                if elem:
                    try:
                        elem.click()
                    except Exception:
                        driver.execute_script("arguments[0].click();", elem)
        # Podemos adicionar um print para indicar o que foi marcado
        print(f"[INFO] QPT {codigo_texto} -> {resposta}")
    # Após marcar tudo, pequena pausa
    time.sleep(0.5)
```

Essa função procura cada pergunta pelo código (por exemplo, procura uma linha de tabela `<tr>` onde apareça "Q001", e dentro dessa linha clica no `<input>` do radio cujo valor (value) seja "Sim" ou "Não" ou "NA" conforme a resposta). Isso pressupõe que no HTML o radio input tenha um value com essas strings (Sim/Não/NA). Se não for o caso (às vezes pode ser value "true"/"false" ou numérico), precisaríamos adaptar. Também, se o código Q001, Q002 etc. aparece visível no texto, o seletor acima funciona. Se não, usamos o texto da pergunta como fallback: por isso tentamos pegar parte do nome (após underline) e procurar. Por exemplo, "ACOMP" de "Q002_ACOMP" para encontrar "Acompanhamento" no texto da pergunta. Esse é um método heurístico; se a tela não mostrar os códigos Q001, então temos que encontrar pelo texto mesmo. Nesse caso, seria melhor ter no YAML ou em algum config os textos exatos das perguntas para buscar. Por simplicidade, estamos tentando com pedaços.

- **Preenchimento de EPI adicional (radios de EPIs adicionais):** Provavelmente uma aba ou seção "EPI" (talvez a primeira parte da aba EPI antes da tabela) onde há perguntas tipo "Necessita cinto de segurança? (Sim/Não)". O plano `epi_radios` fornecido tem chaves correspondentes (Q001_CINTO etc). Podemos fazer similar ao questionário:

```
def preencher_epi_adicional(driver, plano_epi: dict, timeout: float):
    """Preenche a seção de EPI adicional necessário conforme o plano
(radios de EPIs adicionais)."""
    # Essa seção pode estar na mesma aba "EPI" ou separada; vamos
assumir que é na aba "EPI" (parte superior).
    goto_tab(driver, "EPI", timeout)
    for epi_cod, resp in plano_epi.items():
        # Similar lógica: procurar pelo identificador (ex: "Q001" ou
parte do texto "CINTO")
```

```
        codigo = epi_cod.split('_')[0] if '_' in epi_cod else epi_cod
        xpath_radio = f"//div[label[contains(.,'{codigo}')]]//
input[@type='radio' and @value='{resp}']"
        elem = safe_find_element(driver, xpath_radio, 2)
        if not elem:
            # tentar por texto
            trecho = epi_cod.split('_', 1)[1] if '_' in epi_cod else
epi_cod
            alt_xpath = f"//div[label[contains(.,'{trecho}')]]//
input[@type='radio' and @value='{resp}']"
            elem = safe_find_element(driver, alt_xpath, 2)
        if elem:
            try:
                elem.click()
            except Exception:
                driver.execute_script("arguments[0].click();", elem)
            print(f"[INFO] EPI adicional {epi_cod} -> {resp}")
    time.sleep(0.5)
```

Aqui assumimos que cada pergunta de EPI adicional está em um `<div>` com um `<label>` contendo algo como "Q001" ou "CINTO", e um `<input type=radio value="Sim">`. Ajuste o XPath conforme a real estrutura da aba EPI adicional.

- **Preenchimento da Análise Ambiental:** Essa é provavelmente uma lista de perguntas também (ex.: "Há derramamento de óleo?" etc.). Pelo que foi indicado, para trabalho a quente geralmente todas respostas são "Não". De qualquer forma, nosso plano não chega a listar as perguntas de Análise Ambiental (no original era tudo "Não" fixo [14]). Podemos simplesmente marcar todos radios como "Não" nessa aba:

```
def preencher_analise_ambiental(driver, timeout: float):
    """Preenche todas perguntas da Análise Ambiental como 'Não' (padrão
para TQ)."""
    goto_tab(driver, "Análise Ambiental", timeout)
    # Seleciona todos radios de valor "Não" visíveis e clica.
    radios = driver.find_elements(By.XPATH, "//input[@type='radio' and
@value='Não']")
    for radio in radios:
        try:
            radio.click()
        except Exception:
            try:
                driver.execute_script("arguments[0].click();", radio)
            except Exception:
                continue
    print("[INFO] Análise Ambiental: todas respostas marcadas como
'Não'.")
    time.sleep(0.5)
```

- **Preenchimento do APN-1:** Similar ao questionário PT, 20 perguntas de Sim/Não. Nosso plano `apn1` tem chaves "Q001", "Q002", etc., com "Sim"/"Não". Podemos fazer parecido:

```
def preencher_apn1(driver, plano_apn: dict, timeout: float):
    """Preenche o formulário APN-1 conforme o plano (Sim/Não para
questões Q001-Q020)."""
    goto_tab(driver, "APN-1", timeout)
    for codigo, resposta in plano_apn.items():
        xpath_radio = f"//tr[td[contains(.,'{codigo}')]]//
input[@type='radio' and @value='{resposta}']"
        elem = safe_find_element(driver, xpath_radio, 2)
        if not elem:
            # Se código não estiver visível, tenta encontrar alguma
identificação textual... (depende da tela APN-1)
            elem = safe_find_element(driver, f"//
td[contains(.,'{codigo}')]/..//input[@value='{resposta}']", 2)
        if elem:
            try:
                elem.click()
            except Exception:
                driver.execute_script("arguments[0].click();", elem)
            print(f"[INFO] APN-1 {codigo} -> {resposta}")
    time.sleep(0.5)
```

Esse conjunto de funções cobre o preenchimento das abas de perguntas. Agora o módulo `epi.py` para a tabela de EPIs por categoria:

Abra `epi.py` e importe:

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import ElementNotInteractableException
import time
from infra import goto_tab
```

Na aba EPI (tabela), provavelmente há seções por categoria de EPI, cada uma com lista de itens selecionáveis (talvez checkboxes ou múltiplos `<select>`s). Supondo que ao clicar na aba "EPI", já estamos na tela que mostra tanto a parte de EPI adicional (preenchida acima) quanto, abaixo, tabelas de EPIs por categoria.

O código original dedicou `quent4_epi.py` a isso. Provavelmente, a lógica foi: - Obter todos elementos já selecionados na tabela. - Comparar com a lista que deveria estar (nosso plano `epis_cat`). - Clicar para adicionar os que faltam, possivelmente removendo os que não deveriam estar (mas se a PT padrão já tem uns obrigatórios, talvez não precisemos desmarcar nada, apenas garantir que os necessários estejam marcados).

Sem saber a exata UI, faremos algo genérico:

```
def processar_aba_epi(driver, epis_categorias: dict, timeout: float):
    """Marca na aba EPI todos os EPIs por categoria listados em
epis_categorias (dict de categoria -> set de EPIs necessários)."""
    goto_tab(driver, "EPI", timeout)
    for categoria, itens in epis_categorias.items():
        for item in itens:
            # Localiza o item pelo nome e marca (pode ser um checkbox ou
similar)
            xpath_checkbox = f"//td[contains(., '{categoria}')]/..//
label[contains(., '{item}')]/preceding-sibling::input[@type='checkbox']"
            # ^ tentativa: encontra a linha da tabela cuja primeira coluna
(categoria) corresponde, e dentro dessa linha procura um label do item.
            elem = driver.find_elements(By.XPATH, xpath_checkbox)
            if elem:
                try:
                    if not elem[0].is_selected():
                        elem[0].click()
                except Exception:
                    try:
                        driver.execute_script("arguments[0].click();",
elem[0])
                    except Exception as ee:
                        print(f"[WARN] Não foi possível marcar EPI '{item}'
em {categoria}: {ee}")
                else:
                    print(f"[INFO] EPI '{item}' marcado em categoria
{categoria}.")
            else:
                # Talvez o item esteja em uma dropdown ou precise buscar de
outra forma
                alt_xpath = f"//label[contains(.,'{item}')]"
                try:
                    alt_elem = WebDriverWait(driver,
1).until(EC.element_to_be_clickable((By.XPATH, alt_xpath)))
                    alt_elem.click()
                    print(f"[INFO] EPI '{item}' marcado (busca
alternativa).")
                except Exception:
                    print(f"[WARN] EPI '{item}' não encontrado na categoria
{categoria}.")
    time.sleep(0.5)
```

Essa função é bastante especulativa, pois depende de como a tabela está estruturada. Ajuste conforme a realidade: - Pode ser que cada categoria seja uma seção separada com seus próprios checkboxes listados, sem tabela HTML. - Se for assim, talvez procurar diretamente por label do item funcione para todos (ignorando categoria, se único no DOM). - Talvez tenha um botão para adicionar EPI e selecione

de uma lista... (o original `EPIAdicionalProcessor` e `quente4_epi` sugere manipulação um pouco mais complexa). - De qualquer forma, a ideia principal é marcar os itens listados.

Nesse ponto, você precisa usar o conhecimento do sistema APLAT P-18: inspecione a aba EPI manualmente, veja se os EPIs por categoria aparecem como checkboxes marcáveis ou se é uma lista multi-seleção que você clica e abre um menu. Se for uma lista multi-select (como um dropdown), a automação teria que clicar no dropdown, buscar o item pelo texto e selecioná-lo. Nesse caso, a abordagem muda: teríamos de localizar o dropdown referente à categoria e inserir o valor. Seria mais complexo e possivelmente por isso no original dedicaram um módulo inteiro com lógica robusta.

Para fins deste guia, vamos supor que marcar via click no label do item funcione.

6. Integração Final e Testes Completos

Agora que todas as funções principais foram implementadas, vamos integrá-las no fluxo do `aplatquente.py`.

No loop de etapas do script principal, depois de gerar e imprimir o plano, adicionamos as chamadas de preenchimento na ordem correta:

```
from preenchimento import preencher_questionario_pt, preencher_epi_adicional,
preencher_analise_ambiental, preencher_apn1
from epi import processar_aba_epi

# ...dentro do for etapa:
if plano:
    # Preencher Questionário PT
    preencher_questionario_pt(driver, plano["qpt"], args.timeout)
    # Preencher EPI adicional (na aba EPI, parte superior)
    preencher_epi_adicional(driver, plano["epi_radios"], args.timeout)
    # Preencher Análise Ambiental
    preencher_analise_ambiental(driver, args.timeout)
    # Preencher APN-1
    preencher_apn1(driver, plano["apn1"], args.timeout)
    # Preencher tabela de EPIs por categoria
    processar_aba_epi(driver, plano["epis_cat"], args.timeout)
    # Depois de tudo preenchido, confirmar a etapa
    clicar_botao_confirmar_rodape(driver, args.timeout)
    fechar_modal_etapa(driver, args.timeout)
    print(f"[INFO] Etapa {etapa} preenchida e confirmada com sucesso.")
```

Lembre de ter importado `clicar_botao_confirmar_rodape` e `fechar_modal_etapa` do `infra` lá em cima também. Por exemplo:

```
from infra import clicar_botao_confirmar_rodape, fechar_modal_etapa
```

Agora o fluxo é: - Abre etapa, - Verifica se é TQ (você pode implementar a checagem ou confiar no contexto), - Gera plano, - Preenche todas as abas conforme o plano, - Confirma.

Testes manuais do fluxo completo: É hora do teste final: 1. Execute o script para uma etapa de trabalho a quente conhecida:

```
python aplatquente.py --valor 12345 --data 2023-10-01 --use-keyring --user
SEUUSUARIO
```

(ou sem keyring se preferir login manual). 2. Observe o comportamento: o navegador deve abrir, logar, buscar a etapa e você verá ele navegando pelas abas preenchendo campos. No console, serão impressas mensagens [INFO] sobre o que está sendo marcado. 3. Após completar, deve aparecer a mensagem de sucesso e possivelmente o navegador fechar (no nosso código, fechamos só depois de ENTER do usuário no final; você pode remover o `input("Pressione ENTER para encerrar...")` agora que testou, ou mantê-lo para depuração visual).

Verifique no sistema APLAT se a PT ficou preenchida corretamente: - Abra a etapa manualmente no APLAT e confira Questionário PT, EPI etc. Estão de acordo com as respostas esperadas? - Se algo não gravou, veja se nosso código de clique achou o elemento correto. Talvez precisemos ajustar algum XPath. Por exemplo, se o questionário PT não registrou uma resposta, pode ser que o `value` do input não é literalmente "Sim"/"Não" e sim "true"/"false". Nesse caso, mude o XPath para clicar pelo valor correto ou clique no `<label>` ao invés do `<input>`. Poderíamos, por exemplo, modificar `preencher_questionario_pt` para encontrar o `<td>` que contém a opção "Sim" e clicar no label dentro. Isso exige inspeção do HTML específico.

Refine o código de preenchimento conforme necessário: - Adicione waits se alguma aba demora a carregar antes de clicar. Por exemplo, após `goto_tab(driver, "APN-1")`, talvez usar um WebDriverWait para a primeira pergunta estar presente. - Adicione tratamento para *stale elements* em loops (no questionário e APN-1, se clicou um radio e a página re-renderiza aquela parte, pode ocorrer StaleElementReference – se perceber isso, pode envolver uma estratégia de mapear todos elementos antes ou usar small delays). - Use logs para itens não encontrados, e decida se isso deve interromper ou seguir. No nosso code, simplesmente avisa e continua.

A ideia de testar *gradualmente* é crucial: se algo falhar no meio, corrija antes de seguir. Por exemplo, se o script não conseguiu clicar no botão Confirmar (talvez o seletor XPATH_BTN_CONFIRMAR esteja errado ou talvez precise rolar a página), arrume isso e teste de novo apenas a parte final.

Após alguns ajustes, você deve atingir um ponto em que **todo o processo roda sem erros para uma etapa de TQ**, preenchendo e confirmando. Teste com outras etapas (varie descrição, características) para garantir que as regras de contexto cobrem os casos. Se encontrar cenários especiais, ajuste as regex de `montar_contexto` ou as regras de EPIs.

7. Finalizando o Projeto – Organização e Melhores Práticas

Com o projeto recriado e funcionando localmente, podemos considerar alguns *toques finais* para torná-lo mais profissional e preparado para evoluções:

- **Organização do Código:** Se tudo estiver num só diretório, avalie transformar em um pacote Python instalável. Por exemplo, colocar os módulos dentro de um pacote `aplatquente/` e ter um setup.py. Como você mencionou que criará um novo repositório no GitHub, pode ser interessante facilitar a instalação:

- Crie um arquivo `setup.py` ou melhor, um `pyproject.toml` com Poetry/Flit, definindo as dependências (selenium, keyring, pyyaml).

- Defina um entry point para console script, de modo que o usuário possa simplesmente rodar `aplatquente --valor 123 --data 2023-10-01` sem precisar invocar `python aplatquente.py`. Por exemplo, usando `setuptools`, no setup.py:

```
entry_points={'console_scripts': ['aplatquente=aplatquente:main']}
```

  e em `aplatquente.py`, proteger a chamada main. Isso torna a ferramenta mais fácil de usar.

- **Documentação:** Atualize o README (ou crie um novo) descrevendo como usar, como configurar keyring, etc., para que futuros usuários (ou você mesmo no futuro) tenham um guia. Aproveite para listar eventuais limitações ou dicas.

- **Controle de Versão:** No repositório Git, acompanhe os commits de cada funcionalidade implementada. Isso ajuda a reverter se algo quebrar. Você pode estruturar commits mais ou menos seguindo as etapas acima (ex: "Setup project structure", "Implement driver and login", "Implement plan generation", "Implement form filling", etc.).

- **Melhorias Futuras:** Embora seu foco atual seja o funcionamento local e estável, tenha em mente algumas ideias inovadoras:

- **Testes automatizados offline:** Crie amostras de HTML das páginas ou funções que simulam o Selenium (usando talvez `selenium.webdriver.Chrome(options=headless)` ou mesmo requests a páginas de teste) para poder validar a lógica de parsing e preenchimento sem precisar do sistema real a cada mudança. Por exemplo, salvar o código HTML de uma etapa de TQ em arquivo e fazer a função de plano ler daquele HTML usando BeautifulSoup – assim você poderia testar diferentes descrições e características rapidamente.
- **Refinamento de Logs:** Considerar usar o módulo `logging` com níveis (info/debug/warning). Por exemplo, os prints `[DEBUG]` poderiam ser logs de debug e suprimidos em execuções normais, enquanto os `[INFO]` e `[WARN]` aparecem. Isso ajuda quando integrando a ferramenta em pipelines.
- **Interface Gráfica ou Web:** Se quiser ir além, poderia criar uma pequena interface (CLI já temos, mas talvez uma GUI ou integração com algum sistema de filas) para usuários menos técnicos usarem. Não é necessário agora, mas é uma ideia para aumentar o alcance.
- **Robustez do WebDriver:** Embora não priorizado agora, no futuro implementar a checagem automática da versão do Edge e download do driver certo (via script) pouparia trabalho manual. Existem projetos (como *webdriver-manager*) que podem auxiliar.

- **Paralelismo:** Se frequentemente há muitas PTs a processar, poderia rodar múltiplas em paralelo (abrindo vários browsers). No entanto, isso consome mais recurso e SSO talvez conflite. De qualquer forma, mantenha em mente se precisar escalar.

- **Segurança:** Você optou por não focar agora, mas sempre valide que a senha não está exposta. No nosso código, se não usar `--use-keyring`, poderíamos implementar de perguntar a senha via `getpass.getpass()` para não mostrar no console. Isso evita que alguém passe a senha como argumento de linha (o que poderia ser visível em histórico ou gerenciador de tarefas). Fica como possível melhoria.

8. Considerações Finais de Execução

Para concluir, ao recriar do zero, obtivemos um projeto mais modular e configurável: - **Configurações externas** (XPaths, regras de preenchimento) isoladas em `config/`, permitindo mudanças sem alterar código. - **Código organizado** em módulos lógicos: infra, plano, preenchimento, epi, facilitando manutenção. - **Testes passo-a-passo** integrados no desenvolvimento para garantir que cada parte funciona antes de juntar tudo, reduzindo riscos de falhas inesperadas.

Siga esses passos com calma, testando em pequenas partes, e você terá uma nova versão funcional. Qualquer erro que surgir, não hesite em consultar como estava implementado no repositório antigo para entender o motivo. A lógica principal permanece a mesma [11] [15], então usar o código anterior como referência (ou até copiar trechos como fizemos) ajuda a não reinventar a roda.

Boa sorte na implementação! Com essa abordagem didática, você terá uma automação sólida e mais fácil de manter no longo prazo.

---

[1] [2] [3] [4] [5] [6] [7] [11] [14] [15] [16] [19] [20] [21] [24] README.md
https://github.com/shesaidshesaid/aplatquente/blob/1cc674f6721b2d46e2cf129bfb24086a6723fc32/README.md

[8] [9] [10] [12] [13] [18] [35] [45] [46] [47] [48] [49] quent2_plano.py
https://github.com/shesaidshesaid/aplatquente/blob/1cc674f6721b2d46e2cf129bfb24086a6723fc32/quent2_plano.py

[17] [23] [25] [26] [27] [28] [33] [34] [36] [37] [38] [39] [40] [41] [42] [43] [44] quent1_infra.py
https://github.com/shesaidshesaid/aplatquente/blob/1cc674f6721b2d46e2cf129bfb24086a6723fc32/quent1_infra.py

[22] [50] [51] [52] aplatquente.py
https://github.com/shesaidshesaid/aplatquente/blob/1cc674f6721b2d46e2cf129bfb24086a6723fc32/aplatquente.py

[29] [30] [31] [32] Use WebDriver to automate Microsoft Edge - Microsoft Edge Developer documentation | Microsoft Learn
https://learn.microsoft.com/en-us/microsoft-edge/webdriver/