Arquivo: worker_homologacao.py

```
from PyQt6.QtWidgets import *
from PyQt6.QtGui import *
from PyQt6.QtCore import *
import re
import time
import os
import pandas as pd
import pdfplumber

class Worker(QThread):
    processing_complete = pyqtSignal(list)  # Sinal para emitir os dados processados
    update_context_signal = pyqtSignal(str)  # Sinal para atualizar o contexto no diálogo
    progress_signal = pyqtSignal(int)  # Sinal para atualizar a barra de progresso

    tabela_base_pronta = pyqtSignal(pd.DataFrame)

    def __init__(self, pdf_dir, modo="completo", parent=None):
        super().__init__(parent)
        self.pdf_dir = pdf_dir
        self.modo = modo
        self.pdf_path_str = str(pdf_dir)

    def run(self):
        """
        Método que será executado na thread.
        Ele verifica o 'modo' e decide qual tarefa executar.
        """
        try:
            # ROTEADOR: Escolhe a função a ser executada com base no modo
            if self.modo == 'tabela_base':
                self._executar_geracao_tabela_base()

            # O modo 'completo' executa exatamente o seu código original.
            elif self.modo == 'completo':
                # --- INÍCIO DA SUA LÓGICA ORIGINAL ---
                extracted_data = []
                pdf_files = list(self.pdf_dir.glob("*.pdf"))
                total_files = len(pdf_files)

                for index, pdf_file in enumerate(pdf_files):
                    text = self.process_single_pdf(pdf_file)
                    if text:
                        extracted_data.append({
                            "nome_arquivo": pdf_file.name,
                            "text": text
                        })
                         self.update_context_signal.emit(f"Análise do PDF {pdf_file.name} concluída
com êxito.")
                    else:
                                self.update_context_signal.emit(f"Não foi possível obter os dados
```

```
corretamente de {pdf_file.name}.")

                    progress = int((index + 1) / total_files * 100)
                    self.progress_signal.emit(progress)
                    QThread.msleep(100)

                self.processing_complete.emit(extracted_data)
                # --- FIM DA SUA LÓGICA ORIGINAL ---

        except Exception as e:
            self.update_context_signal.emit(f"ERRO CRÍTICO no Worker: {e}")

    def _processar_para_tabela_base(self):
        """
        Extrai dados do TR e dos relatórios de homologação para criar uma planilha base.
        """
        pdf_files = [f for f in os.listdir(self.pdf_path_str) if f.lower().endswith('.pdf')]

        # 1. Identificar o arquivo do Termo de Referência (TR)
        path_tr = None
        for arquivo in pdf_files:
            if "termo" in arquivo.lower() and "referencia" in arquivo.lower():
                path_tr = os.path.join(self.pdf_path_str, arquivo)
                self.update_context_signal.emit(f"Termo de Referência encontrado: {arquivo}")
                break

        if not path_tr:
              self.update_context_signal.emit("ERRO: PDF do Termo de Referência não encontrado na
pasta.")
            return

        # 2. Extrair tabela do Termo de Referência com pdfplumber
        df_tr = pd.DataFrame()
        try:
            self.update_context_signal.emit("Extraindo tabela de itens do TR...")
            with pdfplumber.open(path_tr) as pdf:
                todas_as_tabelas = []
                for i, page in enumerate(pdf.pages):
                     self.progress_signal.emit(int((i + 1) / len(pdf.pages) * 50)) # Progresso até
50%
                    tabelas_pagina = page.extract_tables()
                    if tabelas_pagina:
                        # Pega a primeira tabela encontrada na página e adiciona à lista
                        todas_as_tabelas.extend(tabelas_pagina[0])

            if todas_as_tabelas:
                # Usa a primeira linha como cabeçalho e o resto como dados
                     # Assumindo que as colunas de interesse são as 3 primeiras: Item, Catálogo,
Descrição
                colunas_desejadas = ['item', 'catalogo', 'descricao_detalhada']
                df_tr = pd.DataFrame(todas_as_tabelas[1:], columns=todas_as_tabelas[0])
                df_tr = df_tr.iloc[:, :3] # Seleciona apenas as 3 primeiras colunas
                df_tr.columns = colunas_desejadas
```

```
                self.update_context_signal.emit(f"Sucesso: {len(df_tr)} itens extraídos do TR.")
            else:
                self.update_context_signal.emit("AVISO: Nenhuma tabela encontrada no arquivo TR.")

        except Exception as e:
            self.update_context_signal.emit(f"Falha ao extrair dados do TR: {e}")
            return

        # 3. Extrair a descrição curta dos relatórios de homologação
        dados_homologacao = []
        regex_item = re.compile(r"Item:\s*(\d+)")
        regex_desc = re.compile(r"Descrição:\s*([^\n]+)")

        arquivos_homologacao = [f for f in pdf_files if "homologacao" in f.lower()]
        total_homologacao = len(arquivos_homologacao)

        for index, arquivo in enumerate(arquivos_homologacao):
             self.progress_signal.emit(50 + int((index + 1) / total_homologacao * 50)) # Progresso
de 50% a 100%
            path_arquivo = os.path.join(self.pdf_path_str, arquivo)
            try:
                with pdfplumber.open(path_arquivo) as pdf:
                          texto_completo = "".join([page.extract_text() for page in pdf.pages if
page.extract_text()])

                    num_item = regex_item.search(texto_completo)
                    desc_item = regex_desc.search(texto_completo)

                    if num_item and desc_item:
                        dados_homologacao.append({
                            'item': num_item.group(1).strip(),
                            'descricao': desc_item.group(1).strip()
                        })
            except Exception as e:
                self.update_context_signal.emit(f"AVISO: Não foi possível ler {arquivo}: {e}")

        df_homolog = pd.DataFrame(dados_homologacao)
             self.update_context_signal.emit(f"{len(df_homolog)} descrições curtas extraídas dos
relatórios.")

        # 4. Combinar os DataFrames
        if not df_tr.empty:
            if not df_homolog.empty:
                # Garante que a coluna 'item' seja do mesmo tipo para o merge
                df_tr['item'] = df_tr['item'].astype(str)
                df_homolog['item'] = df_homolog['item'].astype(str)
                df_final = pd.merge(df_tr, df_homolog, on='item', how='left')
                self.update_context_signal.emit("Dados do TR e da Homologação combinados.")
            else:
                df_final = df_tr
                df_final['descricao'] = '' # Adiciona coluna de descrição vazia por consistência
                         self.update_context_signal.emit("AVISO: Nenhum dado de homologação para
combinar.")
```

```
            # Envia o DataFrame final através do novo sinal
            self.tabela_base_pronta.emit(df_final)
        else:
             self.update_context_signal.emit("ERRO: Não foi possível gerar a tabela base pois o TR está vazio.")

        self.progress_signal.emit(100)
    def _processar_modo_completo(self):
        """
        Lógica original do método run, para manter o processamento completo.
        """
        extracted_data = []
        pdf_files = list(self.pdf_dir.glob("*.pdf"))
        total_files = len(pdf_files)

        for index, pdf_file in enumerate(pdf_files):
            text = self.process_single_pdf(pdf_file) # Reutiliza a função de processamento de PDF
            if text:
                extracted_data.append({
                    "nome_arquivo": pdf_file.name,
                    "text": text
                })
                    self.update_context_signal.emit(f"Análise do PDF {pdf_file.name} concluída com êxito.")
            else:
                 self.update_context_signal.emit(f"Não foi possível obter os dados corretamente de {pdf_file.name}.")

            progress = int((index + 1) / total_files * 100)
            self.progress_signal.emit(progress)
            QThread.msleep(100)

        self.processing_complete.emit(extracted_data)


    def process_single_pdf(self, pdf_file):
        # Processa um único arquivo PDF e retorna o texto extraído
        try:
            with pdfplumber.open(pdf_file) as pdf:
                text = ""
                for page in pdf.pages:
                    page_text = page.extract_text()
                    if page_text:
                        text += page_text + "\n"
                    else:
                            self.update_context_signal.emit(f"Aviso: Página de {pdf_file.name} não contém texto extraível.")
                return text
        except Exception as e:
            self.update_context_signal.emit(f"Erro ao processar {pdf_file.name}: {e}")
            return None
```

```
class CustomTreeView(QTreeView):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.cnpj = ""

    def contextMenuEvent(self, event):
        index = self.indexAt(event.pos())
        if not index.isValid():
            return

        item = self.model().itemFromIndex(index)
        if item and " - " in item.text():
            # Usar QTextDocument para extrair texto puro a partir do HTML
            doc = QTextDocument()
            doc.setHtml(item.text())
            plain_text = doc.toPlainText()

            # Extração do CNPJ sem HTML
            if " - " in plain_text:
                self.cnpj = plain_text.split(' - ')[0]

            menu = QMenu(self)
            copyAction = QAction(f"Copiar CNPJ {self.cnpj}", self)
            copyAction.triggered.connect(lambda: self.copy_cnpj(plain_text))
            menu.addAction(copyAction)
            menu.exec(event.globalPos())

    def copy_cnpj(self, text):
        if " - " in text:
            self.cnpj = text.split(' - ')[0]  # Extrai o CNPJ
        QApplication.clipboard().setText(self.cnpj)
         QToolTip.showText(QCursor.pos(), f"CNPJ {self.cnpj} copiado para área de transferência",
self)

    def mousePressEvent(self, event):
        index = self.indexAt(event.pos())
        if index.isValid() and event.button() == Qt.MouseButton.LeftButton:
            # Expande ou colapsa o item clicado
            self.setExpanded(index, not self.isExpanded(index))

            # Se o item foi expandido, expanda também o primeiro nível de subitens
            if self.isExpanded(index):
                model = self.model()
                numRows = model.rowCount(index)
                for row in range(numRows):
                    childIndex = model.index(row, 0, index)
                    self.setExpanded(childIndex, True)
                    # Colapsa todos os subníveis abaixo do primeiro nível
                    self.collapseAllChildren(childIndex)

        super().mousePressEvent(event)

    def collapseAllChildren(self, parentIndex):
```

```
        """Recursivamente colapsa todos os subníveis de um dado índice."""
        model = self.model()
        numRows = model.rowCount(parentIndex)
        for row in range(numRows):
            childIndex = model.index(row, 0, parentIndex)
            self.collapseAllChildren(childIndex)
            self.setExpanded(childIndex, False)

    def update_treeview_with_dataframe(self, dataframe):
        if dataframe is None:
             QMessageBox.critical(self, "Erro", "O DataFrame não está disponível para atualizar a
visualização.")
            return

        creator = ModeloTreeview(self.icons_dir)
        self.model = creator.criar_modelo(dataframe)

        # Verifique se o treeView está inicializado
        if not hasattr(self, 'treeView'):
            self.setup_treeview()

        self.treeView.setModel(self.model)
        # self.treeView.setItemDelegate(HTMLDelegate())
        self.treeView.reset()

    def setup_treeview(self):
        # Verificar se o treeView já existe, e criar se não existir
        if not hasattr(self, 'treeView') or self.treeView is None:
            self.treeView = CustomTreeView()  # Inicializar o treeView se ele não existir

        # Remover ou ocultar a mensagem quando o treeView for exibido
        if hasattr(self, 'message_label') and self.message_label:
            self.message_label.hide()

        # Verificar se o modelo já foi inicializado
        if not hasattr(self, 'model') or self.model is None:
                 self.model = QStandardItemModel()   # Inicializando o modelo se ainda não foi
inicializado

        # Configurar o treeView com o modelo
        self.treeView.setModel(self.model)

        # Verificar se o dynamic_content_layout existe e é válido antes de tentar modificar
        if hasattr(self, 'dynamic_content_layout') and self.dynamic_content_layout is not None:
            # Limpar widgets antigos do layout dinâmico
            while self.dynamic_content_layout.count():
                widget_to_remove = self.dynamic_content_layout.takeAt(0).widget()
                if widget_to_remove is not None:
                    widget_to_remove.deleteLater()

            # Adicionar o treeView ao layout dinâmico
            self.dynamic_content_layout.addWidget(self.treeView)
```

```
        # Configurações adicionais para o treeView
        self.treeView.header().setDefaultAlignment(Qt.AlignmentFlag.AlignCenter)
        self.treeView.setAnimated(True)  # Facilita a visualização da expansão/colapso
        self.treeView.setUniformRowHeights(True)  # Uniformiza a altura das linhas
            self.treeView.setItemsExpandable(True)   # Garantir que o botão para expandir esteja
visível
        self.treeView.setExpandsOnDoubleClick(False)  # Evita a expansão por duplo clique
        self.setup_treeview_styles()

    def setup_treeview_styles(self):
        header = self.treeView.header()
        header.setSectionResizeMode(QHeaderView.ResizeMode.ResizeToContents)

class CustomProgressBar(QProgressBar):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setFont(QFont("Arial", 16))  # Aumentar a fonte do percentual
        self.setTextVisible(False)  # Desativar o texto padrão do QProgressBar

    def paintEvent(self, event):
        super().paintEvent(event)
        painter = QPainter(self)
        painter.setPen(QColor(Qt.GlobalColor.black))
        rect = self.rect()
        font_metrics = self.fontMetrics()
        progress_percent = self.value()
        text = f"{progress_percent}%" if progress_percent >= 0 else ""
        text_width = font_metrics.horizontalAdvance(text)
        text_height = font_metrics.height()
         painter.drawText(int((rect.width() - text_width) / 2), int((rect.height() + text_height) /
2), text)
        painter.end()

class TreeViewWindow(QDialog):

    def __init__(self, dataframe, icons_dir, database_ata_manager, parent=None):
        super().__init__(parent)
        self.setWindowTitle("Resultados do Processamento (Em construção)")
        self.dataframe = dataframe
        self.icons_dir = icons_dir
          self.database_ata_manager = database_ata_manager   # Adiciona o gerenciador do banco de
dados
        self.setMinimumSize(800, 600)

        self.setup_ui()

    def setup_ui(self):
        layout = QVBoxLayout(self)

        # Cria o TreeView e o popula
        self.treeView = CustomTreeView()
        self.populate_treeview()
```

```
        layout.addWidget(self.treeView)
        self.setLayout(layout)

    def populate_treeview(self):
        # Passa o database_ata_manager ao criar o modelo
        creator = ModeloTreeview(self.icons_dir, self.database_ata_manager)
        model = creator.criar_modelo(self.dataframe)
        self.treeView.setModel(model)

class ModeloTreeview:
    def __init__(self, icons_dir, database_ata_manager):
        self.icons = icons_dir
          self.database_ata_manager = database_ata_manager  # Armazena o gerenciador do banco de
dados

    def get_icon_for_cnpj(self, cnpj):
        """Verifica se o CNPJ existe na tabela registro_sicaf e retorna o ícone correspondente."""
        try:
            query = "SELECT 1 FROM registro_sicaf WHERE cnpj = ?"
            result = self.database_ata_manager.execute_query(query, (cnpj,))
            return self.icons["check"] if result else self.icons["alert"]
        except Exception as e:
            print(f"Modelo Treeview Erro ao acessar o banco de dados: {e}")
            return self.icons["alert"]

    def determinar_itens_iguais(self, row, empresa_items):
        empresa_name = str(row['empresa']) if pd.notna(row['empresa']) else ""
        cnpj = str(row['cnpj']) if pd.notna(row['cnpj']) else ""
        situacao = str(row['situacao']) if pd.notna(row['situacao']) else "Não definido"
        is_situacao_only = not empresa_name and not cnpj
        parent_key = f"{situacao}" if is_situacao_only else f"{cnpj} - {empresa_name}".strip(" -")

        if parent_key not in empresa_items:
            parent_item = QStandardItem()
            parent_item.setEditable(False)

            # Determina o ícone usando get_icon_for_cnpj
            if cnpj:
                icon = self.get_icon_for_cnpj(cnpj)
            else:
                 icon = self.icons.get("alert", QIcon())  # Ícone de alerta caso o CNPJ seja vazio
ou ausente

            parent_item.setIcon(icon)
            return parent_key, parent_item

        return parent_key, None

    def update_view(self, view):
        # Força a atualização da view para garantir que os ícones sejam exibidos
        view.viewport().update()

    def criar_modelo(self, dataframe):
```

```
        model, header = self.initializar_modelo(dataframe)
        empresa_items = self.processar_linhas(dataframe, model)
        self.atualizar_contador_cabecalho(empresa_items, model)
        return model

    def initializar_modelo(self, dataframe):
        total_items = len(dataframe)
         situacoes_analizadas = ['Adjudicado e Homologado', 'Fracassado e Homologado', 'Deserto e
Homologado', 'Anulado e Homologado', 'Revogado e Homologado']
        nao_analisados = len(dataframe[~dataframe['situacao'].isin(situacoes_analizadas)])
            header = f"Total de itens da licitação {total_items} | Total de itens analisados
{total_items - nao_analisados} | Total de itens não analisados {nao_analisados}"
        model = QStandardItemModel()
        model.setHorizontalHeaderLabels([header])
        return model, header

    def processar_linhas(self, dataframe, model):
        empresa_items = {}
        for _, row in dataframe.iterrows():
            self.processar_linhas_individualmente(row, model, empresa_items)
        return empresa_items

    def processar_linhas_individualmente(self, row, model, empresa_items):
        parent_key, parent_item = self.determinar_itens_iguais(row, empresa_items)
        if parent_item:
            model.appendRow(parent_item)
            if parent_key not in empresa_items:
                empresa_items[parent_key] = {
                    'item': parent_item,
                    'count': 0,
                    'details_added': False,
                    'items_container': QStandardItem()  # Não defina o texto aqui
                }
                empresa_items[parent_key]['items_container'].setEditable(False)

        # Ajuste para incrementar a contagem em todas as situações
        if parent_key in empresa_items:
            empresa_items[parent_key]['count'] += 1

          self.adicionar_informacao_ao_item(row, empresa_items[parent_key]['item'], empresa_items,
parent_key)

    def adicionar_informacao_ao_item(self, row, parent_item, empresa_items, parent_key):
        font_size = "14px"  # Define o tamanho da fonte
          situacoes_especificas = ['Fracassado e Homologado', 'Deserto e Homologado', 'Anulado e
Homologado', 'Revogado e Homologado']
        situacao = row['situacao']

        # Determinar se a situação é específica ou "Não definido"
        if situacao not in situacoes_especificas and situacao != 'Adjudicado e Homologado':
            situacao = 'Não definido'

        if situacao in situacoes_especificas or situacao == 'Não definido':
```

```
            # Cria um item com informações básicas sem detalhes extras para situações específicas
            item_text = f"Item {row['item']} - {row['descricao']} - {situacao}"
            item_info = QStandardItem(item_text)
            item_info.setEditable(False)
            parent_item.appendRow(item_info)
        else:
            # Processo normal para 'Adjudicado e Homologado'
            pass

            # Adicionando itens específicos da licitação
            self.adicionar_subitens_detalhados(row, empresa_items[parent_key]['items_container'])

        # Atualizar o texto do container com base na contagem de itens
           item_count_text = "Item" if empresa_items[parent_key]['count'] == 1 else "Relação de
itens:"
                           empresa_items[parent_key]['items_container'].setText(f"{item_count_text}
({empresa_items[parent_key]['count']})")

    def atualizar_contador_cabecalho(self, empresa_items, model):
        font_size = "16px"  # Definir o tamanho da fonte para os cabeçalhos dos itens
        for chave_item_pai, empresa in empresa_items.items():
            count = empresa['count']
            # Formatar o texto com HTML para ajustar o tamanho da fonte
               display_text = f"{chave_item_pai} (1 item)" if count == 1 else f"{chave_item_pai}
({count} itens)"
            empresa['item'].setText(display_text)

    def adicionar_detalhes_empresa(self, row, parent_item):
        infos = [
              f"Endereço: {row['endereco']}, CEP: {row['cep']}, Município: {row['municipio']}" if
pd.notna(row['endereco']) else "Endereço: Não informado",
             f"Contato: {row['telefone']} Email: {row['email']}" if pd.notna(row['telefone']) else
"Contato: Não informado",
             f"Responsável Legal: {row['responsavel_legal']}" if pd.notna(row['responsavel_legal'])
else "Responsável Legal: Não informado"
        ]
        for info in infos:
            info_item = QStandardItem(info)
            info_item.setEditable(False)
            parent_item.appendRow(info_item)

    def criar_dados_sicaf_do_item(self, row):
        fields = ['endereco', 'cep', 'municipio', 'telefone', 'email', 'responsavel_legal']
          return [self.criar_detalhe_item(field.capitalize(), row[field]) for field in fields if
pd.notna(row[field])]

    def adicionar_subitens_detalhados(self, row, sub_items_layout):
        item_info_html = f"Item {row['item']} - {row['descricao']} - {row['situacao']}"
        item_info = QStandardItem(item_info_html)
        item_info.setEditable(False)
        sub_items_layout.appendRow(item_info)

        detalhes = [
```

```
            f"Descrição Detalhada: {row['descricao_detalhada']}",
                                    f"Unidade   de   Fornecimento:   {row['unidade']}   Quantidade:
{self.formatar_quantidade(row['quantidade'])} "
                    f"Valor  Estimado:  {self.formatar_brl(row['valor_estimado'])}  Valor  Homologado:
{self.formatar_brl(row['valor_homologado_item_unitario'])} "
                         f"Desconto:  {self.formatar_percentual(row['percentual_desconto'])}  Marca:
{row['marca_fabricante']} Modelo: {row['modelo_versao']}",
        ]

        for detalhe in detalhes:
            detalhe_item = QStandardItem(detalhe)
            detalhe_item.setEditable(False)
            item_info.appendRow(detalhe_item)



    def criar_detalhe_item(self, label, data):
        return QStandardItem(f"<b>{label}:</b> {data if pd.notna(data) else 'Não informado'}")

    def formatar_brl(self, valor):
        try:
            if valor is None:
                return "Não disponível"  # ou outra representação adequada para seu caso de uso
            return f"R$ {float(valor):,.2f}".replace(",", "X").replace(".", ",").replace("X", ".")
        except ValueError:
            return "Valor inválido"

    def formatar_quantidade(self, valor):
        try:
            float_value = float(valor)
            if float_value.is_integer():
                return f"{int(float_value)}"
            else:
                return f"{float_value:.2f}".replace('.', ',')
        except ValueError:
            return "Erro de Formatação"

    def formatar_percentual(self, valor):
        try:
            percent_value = float(valor)
            return f"{percent_value:.2f}%"
        except ValueError:
            return "Erro de Formatação"

import re
import pandas as pd

class WorkerSICAF(QThread):
    processing_complete = pyqtSignal(list)  # Signal to emit list of DataFrames
    update_context_signal = pyqtSignal(str)
    progress_signal = pyqtSignal(int)

    def __init__(self, sicaf_dir, parent=None):
        super().__init__(parent)
```

```
        self.sicaf_dir = sicaf_dir

    def run(self):
        dataframes = []
        pdf_files = list(self.sicaf_dir.glob("*.pdf"))
        total_files = len(pdf_files)

        for index, pdf_file in enumerate(pdf_files):
            text = self.process_single_pdf(pdf_file)
            if text:
                # Extrair dados usando as expressões regulares
                df_sicaf = extrair_dados_sicaf(text)
                df_responsavel = extrair_dados_responsavel(text)

                print(f"Arquivo PDF: {pdf_file.name}")
                print("DataFrame SICAF extraído:")
                print(df_sicaf)
                print("DataFrame Responsável extraído:")
                print(df_responsavel)

                if not df_sicaf.empty or not df_responsavel.empty:
                    # Combinar os DataFrames
                    df_combined = pd.concat([df_sicaf, df_responsavel], axis=1)
                    dataframes.append(df_combined)
                    message = f"Análise do PDF {pdf_file.name} concluída com êxito."
                    self.update_context_signal.emit(message)
                    print(message)
                else:
                    message = f"Nenhum dado extraído de {pdf_file.name}."
                    self.update_context_signal.emit(message)
                    print(message)
            else:
                message = f"Não foi possível obter os dados corretamente de {pdf_file.name}."
                self.update_context_signal.emit(message)
                print(message)

            progress = int((index + 1) / total_files * 100)
            self.progress_signal.emit(progress)
            QThread.msleep(1)  # Simulando tempo de processamento

        # Emitir o sinal com a lista de DataFrames
        self.processing_complete.emit(dataframes)

    def process_single_pdf(self, pdf_file):
        try:
            with pdfplumber.open(pdf_file) as pdf:
                text = ""
                for page in pdf.pages:
                    page_text = page.extract_text()
                    if page_text:
                        text += page_text + "\n"
                    else:
                            warning_message = f"Aviso: Página de {pdf_file.name} não contém texto
```

```
extraível."
                        self.update_context_signal.emit(warning_message)
                        print(warning_message)
                return text
        except Exception as e:
            error_message = f"Erro ao processar {pdf_file.name}: {e}"
            self.update_context_signal.emit(error_message)
            print(error_message)
            return None


# def extrair_dados_sicaf(text):
#     # Definir a expressão regular para extração dos dados
#     dados_sicaf = (
#         r"CNPJ:\s*(?P<cnpj>[\d./-]+)\s*"
#         r"(DUNS®:\s*(?P<duns>[\d]+)\s*)?"
#         r"Razão Social:\s*(?P<empresa>.*?)\s*"
#         r"Nome Fantasia:\s*(?P<nome_fantasia>.*?)\s*"
#         r"Situação do Fornecedor:\s*(?P<situacao_cadastro>.*?)\s*"
#         r"Data de Vencimento do Cadastro:\s*(?P<data_vencimento>\d{2}/\d{2}/\d{4})\s*"
#         r"(Data de Emissão do Certificado:\s*(?P<data_emissao>\d{2}/\d{2}/\d{4})\s*)?"
#         r"(Data de Validade do Certificado:\s*(?P<data_validade>\d{2}/\d{2}/\d{4})\s*)?"
#         r"(Data de Última Atualização:\s*(?P<data_atualizacao>\d{2}/\d{2}/\d{4})\s*)?"
#         r"Dados do Nível.*?Dados para Contato\s*"
#         r"CEP:\s*(?P<cep>[\d.-]+)\s*"
#         r"Endereço:\s*(?P<endereco>.*?)\s*"
#         r"Município / UF:\s*(?P<municipio>.*?)\s*/\s*(?P<uf>.*?)\s*"
#         r"Telefone:\s*(?P<telefone>.*?)\s*"
#         r"E-mail:\s*(?P<email>.*?)\s*"
#         r"Dados do Responsável Legal CPF:\s*(?P<cpf>[\d.-]+)"
#     )

#     # Procurar todos os matches no texto fornecido
#     matches = re.finditer(dados_sicaf, text, re.DOTALL)

#     # Extrair os dados e armazenar em uma lista de dicionários
#     extracted_data = [match.groupdict() for match in matches]

#     # Criar um DataFrame com os dados extraídos
#     df = pd.DataFrame(extracted_data)
#     print("Dados extraídos do texto:")
#     print(df)
#     return df

def extrair_dados_sicaf(texto: str) -> pd.DataFrame:
    dados_sicaf = (
        r"CNPJ:\s*(?P<cnpj>[\d./-]+)\s*"
        r"(?:DUNS®:\s*(?P<duns>[\d]+)\s*)?"
        r"Razão Social:\s*(?P<empresa>.*?)\s*"
        r"Nome Fantasia:\s*(?P<nome_fantasia>.*?)\s*"
        r"Situação do Fornecedor:\s*(?P<situacao_cadastro>.*?)\s*"
        r"Data de Vencimento do Cadastro:\s*(?P<data_vencimento>\d{2}/\d{2}/\d{4})\s*"
        r"Dados do Nível.*?Dados para Contato\s*"
```

```
        r"CEP:\s*(?P<cep>[\d.-]+)\s*"
        r"Endereço:\s*(?P<endereco>.*?)\s*"
        r"Município\s*/\s*UF:\s*(?P<municipio>.*?)\s*/\s*(?P<uf>.*?)\s*"
        r"Telefone:\s*(?P<telefone>.*?)\s*"
        r"E-mail:\s*(?P<email>.*?)\s*"
        r"Dados do Responsável Legal"
    )

    match = re.search(dados_sicaf, texto, re.S)
    if not match:
        return pd.DataFrame()  # Retorna um DataFrame vazio

    data = {key: [value.strip()] if value else [None] for key, value in match.groupdict().items()}

    df = pd.DataFrame(data)
    return df

def extrair_dados_responsavel(texto: str) -> pd.DataFrame:
    dados_responsavel_sicaf = (
        r"Dados do Responsável Legal\s*CPF:\s*(?P<cpf>\d{3}\.\d{3}\.\d{3}-\d{2})\s*Nome:\s*"
        r"(?P<nome>[^\n\r]*?)(?:\s*Dados do Responsável pelo Cadastro|\s*Emitido em:|\s*CPF:|$)"
    )

    match = re.search(dados_responsavel_sicaf, texto, re.S)
    if not match:
        return pd.DataFrame()  # Retorna um DataFrame vazio

    data = {key: [value.strip()] if value else [None] for key, value in match.groupdict().items()}

    df = pd.DataFrame(data)
    return df
```