```
!pip -q install -U transformers accelerate bitsandbytes sentencepiece einops
!pip -q install pdf2image pillow
!apt-get -qq update
!apt-get -qq install -y poppler-utils
```

```
W: Skipping acquire of configured file 'main/source/Sources' as repository 'https://r2u.stat.illinois.edu/ubuntu jammy InRel
```

```
import os
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"
```

```
import io
import gc
import json
import time
import torch
from PIL import Image
from pdf2image import convert_from_bytes

import torchvision.transforms as T
from torchvision.transforms.functional import InterpolationMode

from transformers import AutoTokenizer, AutoModel


# =========================
# ✅ SETTINGS (COLAB SAFE)
# =========================
MODEL_PATH = "OpenGVLab/InternVL2_5-4B-MPO"

generation_config = dict(
    max_new_tokens=512,   # ✅ reduced to avoid long output + memory spike
    do_sample=False       # ✅ deterministic & stable for extraction
)

IMAGENET_MEAN = (0.485, 0.456, 0.406)
IMAGENET_STD  = (0.229, 0.224, 0.225)

DEVICE = "cuda" if torch.cuda.is_available() else "cpu"

# ✅ T4 prefers float16, A100 can do bfloat16
DTYPE = torch.bfloat16 if torch.cuda.is_available() and torch.cuda.get_device_capability()[0] >= 8 else torch.float16


# =========================
# ✅ LOAD MODEL (COLAB VRAM OPTIMIZED)
# =========================
print("Loading tokenizer...")
tokenizer = AutoTokenizer.from_pretrained(MODEL_PATH, trust_remote_code=True)

print("Loading model...")
model = AutoModel.from_pretrained(
    MODEL_PATH,
    trust_remote_code=True,
    torch_dtype=DTYPE,
    load_in_8bit=True,          # ✅ keeps it runnable on T4
    device_map="auto",
    low_cpu_mem_usage=True
).eval()

print("✅ Model loaded on:", DEVICE)


# =========================
# ✅ IMAGE PREPROCESSING
# =========================
def build_transform(input_size):
    transform = T.Compose([
        T.Lambda(lambda img: img.convert("RGB") if img.mode != "RGB" else img),
        T.Resize((input_size, input_size), interpolation=InterpolationMode.BICUBIC),
        T.ToTensor(),
        T.Normalize(mean=IMAGENET_MEAN, std=IMAGENET_STD)
    ])
    return transform


def find_closest_aspect_ratio(aspect_ratio, target_ratios, width, height, image_size):
    best_ratio_diff = float("inf")
    best_ratio = (1, 1)
    area = width * height

    for ratio in target_ratios:
        target_aspect_ratio = ratio[0] / ratio[1]
        ratio_diff = abs(aspect_ratio - target_aspect_ratio)

        if ratio_diff < best_ratio_diff:
            best_ratio_diff = ratio_diff
            best_ratio = ratio
        elif ratio_diff == best_ratio_diff:
            if area > 0.5 * image_size * image_size * ratio[0] * ratio[1]:
                best_ratio = ratio

    return best_ratio


def dynamic_preprocess(image, min_num=1, max_num=6, image_size=448, use_thumbnail=True):
    """
    ✅ Reduced default max_num=6 (from 12) -> prevents token overflow + OOM on T4
    ✅ Reduced default image_size=384 (from 448) -> still high accuracy but lower VRAM
    """
    orig_width, orig_height = image.size
    aspect_ratio = orig_width / orig_height

    target_ratios = set(
        (i, j)
        for n in range(min_num, max_num + 1)
        for i in range(1, n + 1)
        for j in range(1, n + 1)
        if i * j <= max_num and i * j >= min_num
    )

    target_ratios = sorted(target_ratios, key=lambda x: x[0] * x[1])

    target_aspect_ratio = find_closest_aspect_ratio(
        aspect_ratio, target_ratios, orig_width, orig_height, image_size
    )

    target_width  = image_size * target_aspect_ratio[0]
    target_height = image_size * target_aspect_ratio[1]
    blocks = target_aspect_ratio[0] * target_aspect_ratio[1]

    resized_img = image.resize((target_width, target_height))

    processed_images = []
    for i in range(blocks):
        box = (
            (i % (target_width // image_size)) * image_size,
            (i // (target_width // image_size)) * image_size,
            ((i % (target_width // image_size)) + 1) * image_size,
            ((i // (target_width // image_size)) + 1) * image_size
        )
        processed_images.append(resized_img.crop(box))

    if use_thumbnail and len(processed_images) != 1:
        thumbnail_img = image.resize((image_size, image_size))
        processed_images.append(thumbnail_img)

    return processed_images


def load_image(image_file, input_size=448, max_num=6, use_thumbnail=True):
    image = Image.open(image_file).convert("RGB")
    transform = build_transform(input_size=input_size)

    images = dynamic_preprocess(
        image,
        image_size=input_size,
        max_num=max_num,
        use_thumbnail=use_thumbnail
    )

    pixel_values = [transform(img) for img in images]
    pixel_values = torch.stack(pixel_values)
    return pixel_values


# =========================
# ✅ CONFIDENCE CALCULATION
# =========================
def calculate_extraction_confidence(response_dict):
    confidence_values = [
        v for k, v in response_dict.items()
        if k.endswith("_confidence") and isinstance(v, (int, float))
    ]

    if not confidence_values:
        return 0.0

    return round(sum(confidence_values) / len(confidence_values), 2)


# ========================
# ✅ INTERNVL CHAT
# ========================
def internvl_chat(image_pixels, prompt):
    try:
        with torch.no_grad():
            response, history = model.chat(
                tokenizer,
                image_pixels,
                prompt,
                generation_config,
                history=None,
                return_history=True
            )
        return response
    except RuntimeError as e:
        # Catch GPU memory errors clearly
        if "CUDA out of memory" in str(e):
            print("❌ CUDA OOM during InternVL chat")
        else:
            print("❌ Runtime error:", str(e))
        return None
    except Exception as ex:
        print("❌ InternVL Error:", str(ex))
        return None


def ai_analysis(image_pixels, prompt):
    response = internvl_chat(image_pixels, prompt)

    if response:
        cleaned = response.replace("```", "").replace("json", "").strip()

        try:
            parsed = json.loads(cleaned)
            if isinstance(parsed, dict):
                parsed["extraction_confidence"] = calculate_extraction_confidence(parsed)
            return parsed
        except:
            return response

    return None


# ========================
# ✅ PROMPT (same as yours)
# ========================
DEFAULT_EXTRACTION_PROMPT = """
You are an expert at extracting structured data and information from documents including invoices, receipts, forms, tables,
You will be provided images of documents that may contain text, forms, tables, or structured data.

DO NOT make up any information that is not in the given images.
If you cannot find the data, respond with null or "unknown" as applicable.

Extract ALL data from the document with high precision and accuracy:

For documents with tables or item lists (invoices, receipts, purchase orders):
- Extract each row as a complete object
- Preserve all numeric values (prices, quantities, amounts, rates, percentages)
- Include all columns/fields exactly as shown
- Return as a JSON array of objects

For form documents:
- Extract all filled fields and their corresponding values
- Preserve field names and values exactly
- Return as a single JSON object

For text documents:
- Extract key information and structure logically
- Identify and preserve numeric data
```

```
- Maintain field names and values

Pay special attention to:
- Numeric values (prices, quantities, totals, percentages, tax amounts)
- Names and identifiers (e.g., patient names, provider names, product names)
- Dates and amounts
- Table headers and row data
- Currency symbols and units

Important guidelines:
- All numeric values must be numbers
- Preserve original field names exactly as they appear
- Do not trim or modify extracted text
- Include decimal places for prices and percentages
- Handle currency symbols by extracting just the numeric value
- Ensure all extracted data is accurate and complete

Return valid JSON format that matches the document structure.
If the document contains a table or list of items, return a JSON array.
If it's a form or single record, return a JSON object.
""".strip()


# ========================
# ✅ MAIN PDF FUNCTION (SAFE)
# ========================
def extract_text_from_pdf(pdf_file, start_page=1, prompt=None, max_pages=1):
    """
    ✅ max_pages default = 1 (VERY IMPORTANT for T4)
    ✅ process only 1 page -> avoids token overflow & OOM
    """

    pixel_values = None
    model_processing_time = 0

    try:
        # Convert PDF -> images
        images = convert_from_bytes(pdf_file.getvalue())

        if len(images) > 1:
            images = images[start_page - 1:]
        else:
            images = images

        # Limit pages strongly for safety
        images = images[:max_pages]

        pixels_array = []

        for idx, image in enumerate(images):
            img_byte_arr = io.BytesIO()
            image.save(img_byte_arr, format="PNG")
            img_bytes = img_byte_arr.getvalue()

            # ✅ Try high-accuracy settings first
            try_input_size = 448 # Changed from 384
            try_max_num = 6

            # ✅ Convert to pixel tensor
            img_pixel_values = load_image(
                io.BytesIO(img_bytes),
                input_size=try_input_size,
                max_num=try_max_num,
                use_thumbnail=False
            )

            img_pixel_values = img_pixel_values.to(dtype=DTYPE, device=DEVICE)
            pixels_array.append(img_pixel_values)

        pixel_values = torch.cat(pixels_array, dim=0)

        # Prompt
        system_prompt = prompt if prompt else DEFAULT_EXTRACTION_PROMPT

        # Inference
        start_time = time.time()
        resp = ai_analysis(pixel_values, system_prompt)
        model_processing_time = time.time() - start_time

        if resp:
            return {
                "status": "success",
                "highest_confidence_response": resp,
                "execution_time": model_processing_time
            }, 0, model_processing_time

        return {
            "status": "no_response",
            "highest_confidence_response": None,
            "execution_time": model_processing_time
        }, 0, model_processing_time

    except RuntimeError as e:
        # ✅ If GPU OOM, retry with smaller settings automatically
        if "CUDA out of memory" in str(e):
            print("⚠️ OOM detected. Retrying with smaller image settings...")

            try:
                torch.cuda.empty_cache()
                gc.collect()

                images = convert_from_bytes(pdf_file.getvalue())
                images = images[start_page - 1:] if len(images) > 1 else images
                images = images[:max_pages]

                pixels_array = []

                for image in images:
                    img_byte_arr = io.BytesIO()
                    image.save(img_byte_arr, format="PNG")
                    img_bytes = img_byte_arr.getvalue()

                    # ✅ fallback (still decent accuracy)
                    img_pixel_values = load_image(
                        io.BytesIO(img_bytes),
                        input_size=448, # Changed from 320
                        max_num=4,
                        use_thumbnail=False
                    )
                    img_pixel_values = img_pixel_values.to(dtype=DTYPE, device=DEVICE)
                    pixels_array.append(img_pixel_values)

                pixel_values = torch.cat(pixels_array, dim=0)

                system_prompt = prompt if prompt else DEFAULT_EXTRACTION_PROMPT

                start_time = time.time()
                resp = ai_analysis(pixel_values, system_prompt)
                model_processing_time = time.time() - start_time

                if resp:
                    return {
                        "status": "success_fallback",
                        "highest_confidence_response": resp,
                        "execution_time": model_processing_time
                    }, 0, model_processing_time

                return {
                    "status": "no_response_fallback",
                    "highest_confidence_response": None,
                    "execution_time": model_processing_time
                }, 0, model_processing_time

            except Exception as ex2:
                return {
                    "status": "error",
                    "error": f"OOM retry failed: {str(ex2)}"
                }, 0, model_processing_time

        return {
            "status": "error",
            "error": str(e)
        }, 0, model_processing_time

    except Exception as e:
        return {
            "status": "error",
            "error": str(e)
        }, 0, model_processing_time

    finally:
        try:
            del pixel_values
        except:
            pass
        torch.cuda.empty_cache()
        gc.collect()


# ========================
# ✅ SUMMARY FUNCTION (SAFE)
# ========================
def summarize_the_pdf(pdf_file, start_page=1, max_pages=1):
    pixel_values = None

    try:
        images = convert_from_bytes(pdf_file.getvalue())
        images = images[start_page - 1:] if len(images) > 1 else images
        images = images[:max_pages]

        pixels_array = []

        for image in images:
            img_byte_arr = io.BytesIO()
            image.save(img_byte_arr, format="PNG")
            img_bytes = img_byte_arr.getvalue()

            img_pixel_values = load_image(
                io.BytesIO(img_bytes),
                input_size=448, # Changed from 384
                max_num=6,
                use_thumbnail=False
            )
            img_pixel_values = img_pixel_values.to(dtype=DTYPE, device=DEVICE)
            pixels_array.append(img_pixel_values)

        pixel_values = torch.cat(pixels_array, dim=0)

        prompt = "Summarize the given document images clearly and accurately."
        resp = internvl_chat(pixel_values, prompt)
        return resp

    except Exception as ex:
        return str(ex)

    finally:
        try:
            del pixel_values
        except:
            pass
        torch.cuda.empty_cache()
        gc.collect()
```

```
Loading tokenizer...
Loading model...
The `load_in_4bit` and `load_in_8bit` arguments are deprecated and will be removed in the future versions. Please, pass a `E
Loading checkpoint shards: 100%                                   2/2 [00:36<00:00, 17.03s/it]
✅ Model loaded on: cuda
```

```
PDF_PATH = "/content/main_test_file (1).pdf"

# ✅ Read PDF bytes
with open(PDF_PATH, "rb") as f:
    pdf_bytes = f.read()

# ✅ Wrapper because extract_text_from_pdf() expects .getvalue()
class DummyUploadFile:
    def __init__(self, file_bytes, name="test.pdf"):
        self._bytes = file_bytes
        self.name = name

    def getvalue(self):
        return self._bytes

pdf_file = DummyUploadFile(pdf_bytes, name="main_test_file (1).pdf")

# ✅ Run extraction
result, ocr_time, model_time = extract_text_from_pdf(
    pdf_file,
    start_page=1,
    max_pages=1
)

import json
print(json.dumps(result, indent=2, ensure_ascii=False))
```

```
    print(json.dumps(result, indent=2, ensure_ascii=False))


    # ✅ Print output
    print("✅ OCR time:", ocr_time)
    print("✅ Model time:", model_time)
```

```
Setting `pad_token_id` to `eos_token_id`:151645 for open-end generation.
{
  "status": "success",
  "highest_confidence_response": {
    "Claimant's Statement and Authorization": {
      "Instructions": "Complete all applicable parts of this form.",
      "MedicalServicesOutsideUS": "If medical services took place outside the United States, please complete this form along
      "FormSubmissionOptions": {
        "OnlineSubmission": "https://service.worldtrips.com/",
        "PaperForm": {
          "MailTo": {
            "Company": "WorldTrips",
            "BoxNo": 2005,
            "Address": "Farming Hills, MI 48333-2005",
            "Country": "U.S.A"
          }
        }
      },
      "QuestionsOrGuidance": "https://www.worldtrips.com/claims-resource-center. Toll-free: 800-605-2282 in the U.S. or coll
      "PartA": {
        "ClaimantInformation": {
          "Full Name": "Ilyas Malik",
          "Gender": "Male",
          "Date of Birth": "07/03/2008",
          "Current Mailing Address": {
            "Street": "69 Sylvester Avenue",
            "City": "Winchester",
            "State": "MA",
            "Postal Code": "01890",
            "Country": "USA"
          },
          "Primary Telephone": "245549351",
          "Secondary Telephone": null,
          "Email Address": "razmalik@gmail.com",
          "Policy Number or Certificate Number": "120392469",
          "Citizenship": "USA",
          "Home Country": "USA",
          "Countries Visited": [
            "England",
            "Turkey"
          ]
        }
      }
    },
    "extraction_confidence": 0.0
  },
  "execution_time": 97.6566309928894
}
✅ OCR time: 0
✅ Model time: 97.6566309928894
```