Prefect User Guide

Joaquín Urruti

2026-01-04

Table of contents

Prefect Workflows - Guía de Uso

Bienvenido a la guía completa de uso de Prefect en Docker para el proyecto Espartina. Esta guía te enseñará desde conceptos básicos hasta configuraciones avanzadas.

Tabla de Contenidos

8. Trabajar con Outputs
9. Configuración Avanzada
10. Ejemplos Prácticos
11. Best Practices

Introducción a Prefect

Prefect es una plataforma moderna de orquestación de workflows que te permite:

- **Orquestar** pipelines de datos complejos
- **Programar** ejecuciones automáticas con cron
- **Monitorear** el estado de tus workflows en tiempo real
- **Recibir alertas** cuando algo falla
- **Reintentar** automáticamente tareas fallidas
- **Registrar logs** centralizados de todas tus ejecuciones

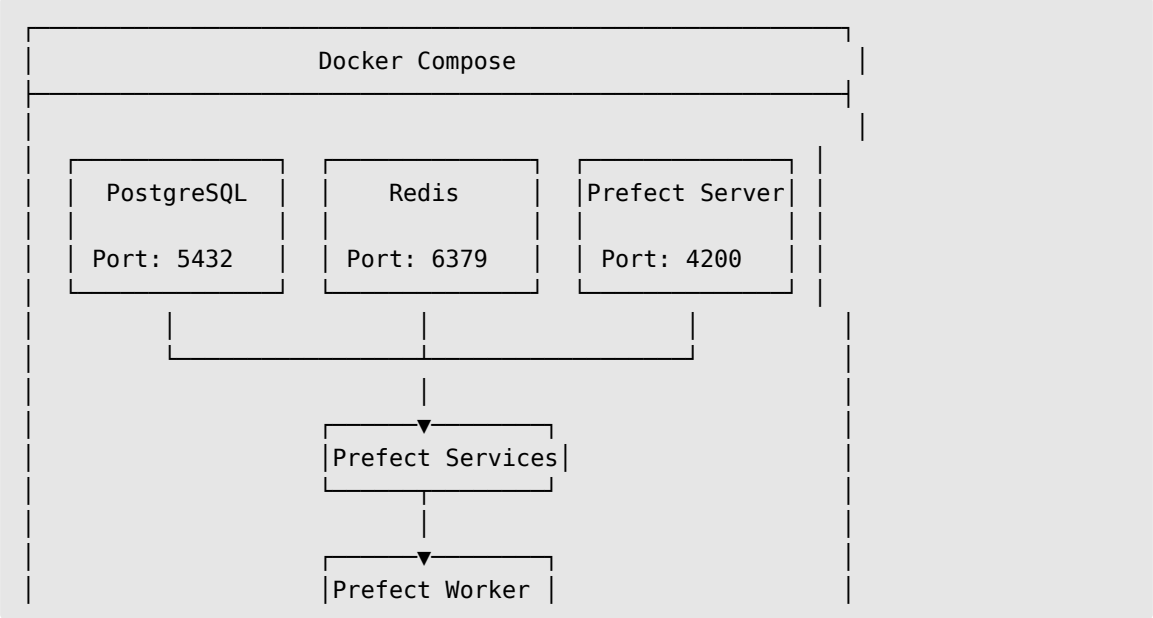
¿Por qué Prefect?

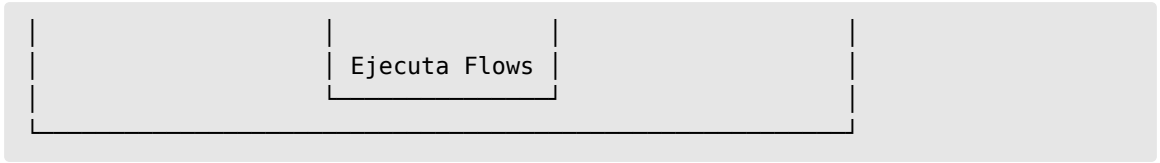
- **Simplicidad:** Escribe código Python normal, Prefect se encarga del resto
- **Observabilidad:** UI moderna para visualizar y debuggear
- **Confiabilidad:** Sistema de reintentos, timeouts y manejo de errores
- **Escalabilidad:** Desde desarrollo local hasta producción distribuida

---

Arquitectura del Proyecto

Componentes de Docker





Persistencia de Datos

```
Host (tu computadora)          Docker Containers
─────────────────────          ─────────────────

./data/postgres/        ←→     PostgreSQL Data
./data/redis/           ←→     Redis Data
./logs/server/          ←→     Prefect Server Logs
./logs/services/        ←→     Prefect Services Logs
./logs/worker/          ←→     Prefect Worker Logs
./outputs/              ←→     Task Outputs
./scripts/              ←→     Python Workflows
```

Conceptos Fundamentales

1. Tasks (Tareas)

Una **task** es una unidad de trabajo individual. Es una función Python decorada con `@task`.

```python
from prefect import task

@task
def extraer_datos(url: str):
    """Extrae datos de una API"""
    # Tu código aquí
    return datos
```

Características: - Pueden recibir parámetros - Retornan valores - Se pueden reintentar automáticamente - Tienen logging integrado - Se ejecutan de forma independiente

2. Flows (Flujos)

Un **flow** es una colección de tasks orquestadas. Es una función Python decorada con `@flow`.

```python
from prefect import flow, task

@task
def tarea_1():
    return "resultado 1"
```

```python
@task
def tarea_2(input_data):
    return f"procesado: {input_data}"

@flow
def mi_workflow():
    resultado = tarea_1()
    final = tarea_2(resultado)
    return final
```

Características: - Orquestan múltiples tasks - Pueden llamar a otros flows (subflows) - Manejan el estado y las dependencias - Se pueden deployar y programar

3. Deployments

Un **deployment** es una configuración que le dice a Prefect cómo y cuándo ejecutar un flow.

```python
if __name__ == "__main__":
    mi_workflow.deploy(
        name="mi-deployment",
        work_pool_name="local-pool",
        cron="0 9 * * *",  # Todos los días a las 9 AM
        tags=["producción", "diario"]
    )
```

4. Work Pools y Workers

- **Work Pool:** Un grupo lógico de workers
- **Worker:** El proceso que ejecuta los flows

En nuestra configuración: - Work Pool: `local-pool` - Worker Type: `process` (ejecuta en procesos separados)

Crear tu Primer Workflow

Paso 1: Crear el Archivo Python

Crea un nuevo archivo en `scripts/mi_primer_flow.py`:

```python
from prefect import flow, task
from prefect.logging import get_run_logger
import datetime

@task
def saludar(nombre: str):
    logger = get_run_logger()
```

```
    mensaje = f"¡Hola {nombre}!"
    logger.info(mensaje)
    return mensaje

@task
def obtener_timestamp():
    logger = get_run_logger()
    ahora = datetime.datetime.now()
    logger.info(f"Timestamp: {ahora}")
    return ahora

@flow(name="Mi Primer Flow")
def mi_primer_flow(nombre: str = "Mundo"):
    """
    Un flow simple que saluda y muestra la hora
    """
    logger = get_run_logger()
    logger.info("Iniciando mi primer flow")

    # Ejecutar tasks
    saludo = saludar(nombre)
    timestamp = obtener_timestamp()

    logger.info("Flow completado exitosamente")
    return {"saludo": saludo, "timestamp": timestamp}

if __name__ == "__main__":
    # Ejecutar localmente para testing
    mi_primer_flow(nombre="Espartina")
```

Paso 2: Ejecutar Localmente

Prueba tu flow directamente:

```
docker compose exec prefect-worker python scripts/mi_primer_flow.py
```

Deberías ver los logs en la consola.

Paso 3: Deployar el Flow

Modifica el archivo para agregar deployment:

```
if __name__ == "__main__":
    mi_primer_flow.deploy(
        name="mi-primer-deployment",
        work_pool_name="local-pool",
        cron="*/5 * * * *",  # Cada 5 minutos
```

```
        tags=["tutorial", "básico"]
    )
```

Ejecuta el deployment:

```
docker compose exec prefect-worker python scripts/mi_primer_flow.py
```

Paso 4: Verificar en la UI

1. Abre http://localhost:4200
2. Ve a “Deployments”
3. Deberías ver “mi-primer-deployment”
4. Ve a “Flow Runs” para ver las ejecuciones

Trabajar con Tasks

Configuración de Tasks

Las tasks soportan muchas configuraciones:

```
from prefect import task
from prefect.tasks import task_input_hash
from datetime import timedelta

@task(
    name="Procesar Datos",
    description="Procesa datos de entrada y retorna resultados",
    tags=["procesamiento", "datos"],
    retries=3,                    # Reintentar 3 veces si falla
    retry_delay_seconds=60,       # Esperar 60s entre reintentos
    timeout_seconds=300,          # Timeout de 5 minutos
    cache_key_fn=task_input_hash, # Cachear resultados basados en inputs
    cache_expiration=timedelta(hours=1),  # Cache válido por 1 hora
    log_prints=True,              # Capturar prints como logs
)
def procesar_datos(datos: list):
    # Procesar datos
    resultado = []
    for item in datos:
        # Tu lógica aquí
        resultado.append(item * 2)
    return resultado
```

Tasks con Manejo de Errores

```python
from prefect import task, flow
from prefect.logging import get_run_logger
import requests

@task(retries=2, retry_delay_seconds=30)
def fetch_api_data(url: str):
    logger = get_run_logger()
    try:
        response = requests.get(url, timeout=10)
        response.raise_for_status()
        logger.info(f"✓ Datos obtenidos de {url}")
        return response.json()
    except requests.RequestException as e:
        logger.error(f"✗ Error al obtener datos: {e}")
        raise

@task
def procesar_datos_api(datos: dict):
    logger = get_run_logger()
    # Procesar los datos
    resultado = datos.get("results", [])
    logger.info(f"Procesados {len(resultado)} items")
    return resultado

@flow
def pipeline_api():
    datos = fetch_api_data("https://api.ejemplo.com/datos")
    if datos:
        return procesar_datos_api(datos)
```

Tasks Paralelas

Prefect ejecuta tasks en paralelo automáticamente cuando es posible:

```python
from prefect import flow, task
import time

@task
def procesar_chunk(chunk_id: int, data: list):
    time.sleep(2)  # Simula procesamiento
    return f"Chunk {chunk_id}: {len(data)} items procesados"

@flow
def procesamiento_paralelo():
    """
    Procesa múltiples chunks en paralelo
    """
    chunks = [
```

```
        [1, 2, 3, 4, 5],
        [6, 7, 8, 9, 10],
        [11, 12, 13, 14, 15],
        [16, 17, 18, 19, 20]
    ]

    # Estas tasks se ejecutan en paralelo
    resultados = []
    for i, chunk in enumerate(chunks):
        resultado = procesar_chunk.submit(i, chunk)  # .submit() = async
        resultados.append(resultado)

    # Esperar a que todas terminen
    return [r.result() for r in resultados]
```

Deployments y Scheduling

Tipos de Schedules

1. Cron Schedule

```
@flow
def mi_flow():
    pass

if __name__ == "__main__":
    mi_flow.deploy(
        name="cron-example",
        work_pool_name="local-pool",
        cron="0 9 * * 1-5"  # Lunes a Viernes a las 9 AM
    )
```

Ejemplos de Cron: - `"0 * * * *"` - Cada hora - `"*/15 * * * *"` - Cada 15 minutos - `"0 9,17 * * *"` - A las 9 AM y 5 PM - `"0 0 * * 0"` - Domingos a medianoche - `"30 2 1 * *"` - Primer día del mes a las 2:30 AM

2. Interval Schedule

```
from prefect.client.schemas.schedules import IntervalSchedule
from datetime import timedelta

if __name__ == "__main__":
    mi_flow.deploy(
        name="interval-example",
        work_pool_name="local-pool",
```

```
        interval=timedelta(hours=2)  # Cada 2 horas
    )
```

3. Sin Schedule (Manual)

```
if __name__ == "__main__":
    mi_flow.deploy(
        name="manual-example",
        work_pool_name="local-pool"
        # Sin cron ni interval = solo ejecución manual
    )
```

Deployment con Parámetros

```
from prefect import flow

@flow
def procesar_archivo(archivo: str, formato: str = "csv"):
    # Tu código aquí
    pass

if __name__ == "__main__":
    procesar_archivo.deploy(
        name="procesar-datos",
        work_pool_name="local-pool",
        parameters={
            "archivo": "/app/outputs/datos.csv",
            "formato": "csv"
        },
        cron="0 10 * * *"
    )
```

Múltiples Deployments del Mismo Flow

Puedes tener diferentes schedules para el mismo flow:

```
if __name__ == "__main__":
    # Deployment 1: Cada hora en horario laboral
    mi_flow.deploy(
        name="horario-laboral",
        work_pool_name="local-pool",
        cron="0 9-18 * * 1-5",
        parameters={"modo": "incremental"}
    )

    # Deployment 2: Diario a medianoche (full refresh)
    mi_flow.deploy(
```

```
        name="refresh-nocturno",
        work_pool_name="local-pool",
        cron="0 0 * * *",
        parameters={"modo": "full"}
    )
```

Logging y Monitoreo

Uso de Loggers

```python
from prefect import flow, task
from prefect.logging import get_run_logger

@task
def procesar_datos(items: list):
    logger = get_run_logger()

    logger.debug("Iniciando procesamiento")
    logger.info(f"Procesando {len(items)} items")

    errores = 0
    for i, item in enumerate(items):
        try:
            # Procesar item
            resultado = item * 2
            logger.debug(f"Item {i}: {item} → {resultado}")
        except Exception as e:
            errores += 1
            logger.error(f"Error en item {i}: {e}")

    if errores > 0:
        logger.warning(f"Se encontraron {errores} errores")
    else:
        logger.info("✓ Procesamiento completado sin errores")

    return len(items) - errores

@flow
def mi_pipeline():
    logger = get_run_logger()
    logger.info("=== Iniciando Pipeline ===")

    datos = [1, 2, 3, 4, 5]
    procesados = procesar_datos(datos)

    logger.info(f"=== Pipeline Completado: {procesados} items ===")
```

Niveles de Logging

Los logs se guardan automáticamente en: - **Contenedor:** `/root/.prefect/logs/` - **Host:** `./logs/worker/`

Niveles disponibles: - `DEBUG` - Información detallada para debugging - `INFO` - Información general del flujo - `WARNING` - Advertencias que no detienen la ejecución - `ERROR` - Errores que afectan la tarea - `CRITICAL` - Errores críticos del sistema

Ver Logs

Desde Docker

```
# Logs del worker en tiempo real
docker compose logs -f prefect-worker

# Logs del servidor
docker compose logs -f prefect-server

# Logs de un período específico
docker compose logs --since 1h prefect-worker
```

Desde la UI

1. Ve a http://localhost:4200
2. Navega a “Flow Runs”
3. Haz clic en cualquier run
4. Ve a la pestaña “Logs”

Desde el Sistema de Archivos

```
# Ver logs más recientes
tail -f logs/worker/*.log

# Buscar errores
grep "ERROR" logs/worker/*.log

# Ver logs de una fecha específica
ls -lh logs/worker/
```

Trabajar con Outputs

Guardar Outputs en Archivos

```
from prefect import flow, task
from prefect.logging import get_run_logger
import json
import csv
```

```python
from pathlib import Path
from datetime import datetime

@task
def guardar_json(data: dict, nombre_archivo: str):
    """Guarda datos en formato JSON"""
    logger = get_run_logger()

    # Crear path con timestamp
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filepath = Path(f"/app/outputs/{nombre_archivo}_{timestamp}.json")

    # Guardar archivo
    with open(filepath, 'w', encoding='utf-8') as f:
        json.dump(data, f, indent=2, ensure_ascii=False)

    logger.info(f"✓ Archivo guardado: {filepath}")
    return str(filepath)

@task
def guardar_csv(data: list, nombre_archivo: str):
    """Guarda datos en formato CSV"""
    logger = get_run_logger()

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filepath = Path(f"/app/outputs/{nombre_archivo}_{timestamp}.csv")

    if not data:
        logger.warning("No hay datos para guardar")
        return None

    # Obtener headers de la primera fila
    headers = data[0].keys()

    with open(filepath, 'w', newline='', encoding='utf-8') as f:
        writer = csv.DictWriter(f, fieldnames=headers)
        writer.writeheader()
        writer.writerows(data)

    logger.info(f"✓ CSV guardado: {filepath} ({len(data)} filas)")
    return str(filepath)

@task
def guardar_reporte_txt(stats: dict, nombre_archivo: str):
    """Guarda un reporte de texto"""
    logger = get_run_logger()

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
```

```
    filepath = Path(f"/app/outputs/{nombre_archivo}_{timestamp}.txt")

    with open(filepath, 'w', encoding='utf-8') as f:
        f.write("=" * 60 + "\n")
        f.write(f"REPORTE GENERADO: {datetime.now()}\n")
        f.write("=" * 60 + "\n\n")

        for key, value in stats.items():
            f.write(f"{key}: {value}\n")

    logger.info(f"✓ Reporte guardado: {filepath}")
    return str(filepath)

@flow
def pipeline_con_outputs():
    """Flow que genera múltiples outputs"""
    logger = get_run_logger()

    # Generar datos de ejemplo
    datos = {
        "timestamp": str(datetime.now()),
        "registros_procesados": 1000,
        "errores": 5,
        "tasa_exito": 99.5
    }

    datos_tabla = [
        {"id": 1, "nombre": "Item 1", "valor": 100},
        {"id": 2, "nombre": "Item 2", "valor": 200},
        {"id": 3, "nombre": "Item 3", "valor": 300}
    ]

    stats = {
        "Total Items": len(datos_tabla),
        "Suma Total": sum(item["valor"] for item in datos_tabla),
       "Promedio": sum(item["valor"] for item in datos_tabla) / len(datos_tabla)
    }

    # Guardar en diferentes formatos
    json_file = guardar_json(datos, "resultados")
    csv_file = guardar_csv(datos_tabla, "tabla_resultados")
    txt_file = guardar_reporte_txt(stats, "reporte_estadisticas")

    logger.info(f"✓ Outputs generados: JSON, CSV, TXT")

    return {
        "json": json_file,
        "csv": csv_file,
```

```
        "txt": txt_file
    }
```

Acceder a los Outputs

Los archivos se guardan en `/app/outputs/` dentro del contenedor, que está mapeado a `./outputs/` en tu host:

```bash
# Ver archivos generados
ls -lh outputs/

# Ver contenido de un JSON
cat outputs/resultados_*.json | jq .

# Ver CSV
cat outputs/tabla_resultados_*.csv
```

Configuración Avanzada

Variables de Entorno en Flows

```python
from prefect import flow, task
import os

@task
def usar_variables_entorno():
    api_key = os.getenv("API_KEY", "default_key")
    env = os.getenv("ENVIRONMENT", "development")
    return f"Env: {env}, API: {api_key[:5]}..."

@flow
def flow_con_env():
    return usar_variables_entorno()
```

Para agregar variables de entorno, edita `docker-compose.yml`:

```yaml
prefect-worker:
  environment:
    PREFECT_API_URL: http://prefect-server:4200/api
    API_KEY: "tu_api_key_aqui"
    ENVIRONMENT: "production"
```

Secrets y Blocks

Para datos sensibles, usa Prefect Blocks:

```python
from prefect.blocks.system import Secret

# Crear secret (hacer una sola vez)
secret = Secret(value="mi_password_secreto")
secret.save("mi-db-password")

# Usar en un flow
@task
def conectar_db():
    from prefect.blocks.system import Secret
    password = Secret.load("mi-db-password").get()
    # Usar password
```

Notificaciones

Configurar notificaciones cuando un flow falla:

```python
from prefect import flow
from prefect.events import emit_event

@flow
def flow_con_notificaciones():
    try:
        # Tu código aquí
        resultado = procesar_datos()

        # Emitir evento de éxito
        emit_event(
            event="flow.completado",
            resource={"prefect.resource.id": "mi_flow"}
        )

        return resultado
    except Exception as e:
        # Emitir evento de error
        emit_event(
            event="flow.error",
            resource={"prefect.resource.id": "mi_flow"},
            payload={"error": str(e)}
        )
        raise
```

Ejemplos Prácticos

Ejemplo 1: ETL Simple

```python
from prefect import flow, task
from prefect.logging import get_run_logger
import requests
import json
from datetime import datetime

@task(retries=3, retry_delay_seconds=60)
def extraer_datos(api_url: str):
    """Extract: Obtener datos de una API"""
    logger = get_run_logger()
    logger.info(f"Extrayendo datos de {api_url}")

    response = requests.get(api_url, timeout=30)
    response.raise_for_status()
    datos = response.json()

    logger.info(f"✓ {len(datos)} registros extraídos")
    return datos

@task
def transformar_datos(datos: list):
    """Transform: Limpiar y transformar datos"""
    logger = get_run_logger()
    logger.info("Transformando datos")

    datos_limpios = []
    for item in datos:
        # Ejemplo de transformación
        transformado = {
            "id": item.get("id"),
            "nombre": item.get("name", "").upper(),
            "valor": float(item.get("value", 0)),
            "procesado_en": datetime.now().isoformat()
        }
        datos_limpios.append(transformado)

    logger.info(f"✓ {len(datos_limpios)} registros transformados")
    return datos_limpios

@task
def cargar_datos(datos: list, archivo_salida: str):
    """Load: Guardar datos procesados"""
    logger = get_run_logger()
    logger.info(f"Cargando datos a {archivo_salida}")

    filepath = f"/app/outputs/{archivo_salida}"
    with open(filepath, 'w') as f:
        json.dump(datos, f, indent=2)
```

```
    logger.info(f"✓ Datos cargados exitosamente")
    return filepath

@flow(name="ETL Pipeline")
def etl_pipeline():
    """Pipeline ETL completo"""
    logger = get_run_logger()
    logger.info("=== Iniciando ETL Pipeline ===")

    # Extract
    datos_crudos = extraer_datos("https://jsonplaceholder.typicode.com/users")

    # Transform
    datos_transformados = transformar_datos(datos_crudos)

    # Load
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    archivo = cargar_datos(datos_transformados, f"etl_output_{timestamp}.json")

    logger.info("=== ETL Pipeline Completado ===")
    return archivo

if __name__ == "__main__":
    # Deploy con schedule diario
    etl_pipeline.deploy(
        name="etl-diario",
        work_pool_name="local-pool",
        cron="0 2 * * *",  # 2 AM todos los días
        tags=["etl", "producción"]
    )
```

Ejemplo 2: Web Scraping

```
from prefect import flow, task
from prefect.logging import get_run_logger
import requests
from bs4 import BeautifulSoup
import csv
from datetime import datetime

@task(retries=2)
def scrape_pagina(url: str):
    """Scrapea una página web"""
    logger = get_run_logger()
    logger.info(f"Scrapeando: {url}")

    response = requests.get(url, timeout=30)
```

```
    response.raise_for_status()

    soup = BeautifulSoup(response.content, 'html.parser')

    # Ejemplo: extraer títulos
    titulos = []
    for elemento in soup.find_all('h2'):
        titulos.append({
            "titulo": elemento.text.strip(),
            "url": url,
            "fecha_scrape": datetime.now().isoformat()
        })

    logger.info(f"✓ {len(titulos)} elementos encontrados")
    return titulos

@task
def guardar_scrape_results(datos: list):
    """Guarda resultados del scraping"""
    logger = get_run_logger()

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filepath = f"/app/outputs/scrape_results_{timestamp}.csv"

    if datos:
        with open(filepath, 'w', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=datos[0].keys())
            writer.writeheader()
            writer.writerows(datos)

        logger.info(f"✓ Resultados guardados: {filepath}")

    return filepath

@flow
def web_scraping_flow(urls: list):
    """Flow de web scraping"""
    logger = get_run_logger()
    logger.info(f"Iniciando scraping de {len(urls)} URLs")

    todos_los_datos = []
    for url in urls:
        datos = scrape_pagina(url)
        todos_los_datos.extend(datos)

    archivo = guardar_scrape_results(todos_los_datos)
    logger.info(f"✓ Scraping completado: {len(todos_los_datos)} items")
```

```
    return archivo
```

Ejemplo 3: Procesamiento de Archivos

```
from prefect import flow, task
from prefect.logging import get_run_logger
import pandas as pd
from pathlib import Path

@task
def leer_archivo_csv(filepath: str):
    """Lee un archivo CSV"""
    logger = get_run_logger()
    logger.info(f"Leyendo archivo: {filepath}")

    df = pd.read_csv(filepath)
    logger.info(f"✓ {len(df)} filas leídas")

    return df

@task
def analizar_datos(df: pd.DataFrame):
    """Realiza análisis de datos"""
    logger = get_run_logger()
    logger.info("Analizando datos")

    analisis = {
        "total_filas": len(df),
        "total_columnas": len(df.columns),
        "columnas": list(df.columns),
        "estadisticas": df.describe().to_dict()
    }

    logger.info(f"✓ Análisis completado")
    return analisis

@task
def generar_reporte(analisis: dict):
    """Genera reporte del análisis"""
    logger = get_run_logger()

    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    filepath = f"/app/outputs/reporte_analisis_{timestamp}.txt"

    with open(filepath, 'w') as f:
        f.write("REPORTE DE ANÁLISIS\n")
        f.write("=" * 50 + "\n\n")
        f.write(f"Total Filas: {analisis['total_filas']}\n")
```

```
        f.write(f"Total Columnas: {analisis['total_columnas']}\n")
        f.write(f"\nColumnas: {', '.join(analisis['columnas'])}\n")

    logger.info(f"✓ Reporte guardado: {filepath}")
    return filepath

@flow
def procesamiento_archivos_flow(archivo_input: str):
    """Procesa archivo y genera reporte"""
    logger = get_run_logger()
    logger.info("=== Iniciando Procesamiento ===")

    df = leer_archivo_csv(archivo_input)
    analisis = analizar_datos(df)
    reporte = generar_reporte(analisis)

    logger.info("=== Procesamiento Completado ===")
    return reporte
```

Best Practices

1. Estructura de Código

```
# ✓ BIEN: Separar concerns
@task
def extraer():
    pass

@task
def transformar():
    pass

@task
def cargar():
    pass

@flow
def etl():
    data = extraer()
    clean = transformar(data)
    cargar(clean)

# ✗ MAL: Todo en un solo task
@task
def hacer_todo():
```

```
    # Extraer, transformar, cargar todo junto
    pass
```

2. Logging Apropiado

```
# ✓ BIEN: Logging informativo
@task
def procesar(items: list):
    logger = get_run_logger()
    logger.info(f"Procesando {len(items)} items")

    for i, item in enumerate(items):
        logger.debug(f"Procesando item {i}")
        # proceso

    logger.info("✓ Procesamiento completado")

# ✗ MAL: Logging excesivo o ausente
@task
def procesar(items: list):
    for item in items:
        print(f"Item: {item}")  # No usar print
```

3. Manejo de Errores

```
# ✓ BIEN: Manejo específico de errores
@task(retries=3)
def task_con_manejo():
    logger = get_run_logger()
    try:
        # código
        pass
    except SpecificException as e:
        logger.error(f"Error específico: {e}")
        raise
    except Exception as e:
        logger.error(f"Error inesperado: {e}")
        raise

# ✗ MAL: Silenciar errores
@task
def task_malo():
    try:
        # código
        pass
    except:
        pass  # ¡No hacer esto!
```

4. Parametrización

```python
# ✓ BIEN: Usar parámetros
@flow
def pipeline(fecha: str, modo: str = "incremental"):
    # Flexible y reusable
    pass

# ✗ MAL: Hardcodear valores
@flow
def pipeline():
    fecha = "2024-01-01"  # Hardcoded
    modo = "full"
```

5. Documentación

```python
# ✓ BIEN: Documentar flows y tasks
@task
def procesar_datos(datos: list) -> dict:
    """
    Procesa una lista de datos y retorna estadísticas.

    Args:
        datos: Lista de diccionarios con datos a procesar

    Returns:
        Dict con estadísticas de procesamiento

    Raises:
        ValueError: Si los datos están vacíos
    """
    if not datos:
        raise ValueError("Datos vacíos")

    # procesamiento
    return {"procesados": len(datos)}
```

Comandos Útiles de Referencia

Gestión de Flows

```bash
# Listar flows
docker compose exec prefect-server prefect flow ls

# Ver detalles de un flow
docker compose exec prefect-server prefect flow inspect "nombre-flow"
```

```
# Listar runs recientes
docker compose exec prefect-server prefect flow-run ls --limit 10
```

Gestión de Deployments

```
# Listar deployments
docker compose exec prefect-server prefect deployment ls

# Ver detalles de un deployment
docker compose exec prefect-server prefect deployment inspect "flow-name/deployment-name"

# Ejecutar un deployment manualmente
docker compose exec prefect-server prefect deployment run "flow-name/deployment-name"

# Pausar un deployment
docker compose exec prefect-server prefect deployment pause "flow-name/deployment-name"

# Reanudar un deployment
docker compose exec prefect-server prefect deployment resume "flow-name/deployment-name"
```

Gestión de Work Pools

```
# Listar work pools
docker compose exec prefect-server prefect work-pool ls

# Ver detalles de un work pool
docker compose exec prefect-server prefect work-pool inspect local-pool

# Pausar un work pool
docker compose exec prefect-server prefect work-pool pause local-pool
```

Recursos Adicionales

Documentación Oficial

- **Prefect Docs:** https://docs.prefect.io
- **API Reference:** https://docs.prefect.io/api-ref/
- **Concepts:** https://docs.prefect.io/concepts/

Comunidad

- **Slack:** https://prefect.io/slack
- **GitHub:** https://github.com/PrefectHQ/prefect

- **Discourse:** https://discourse.prefect.io

Tutoriales
- **Prefect Recipes:** https://docs.prefect.io/recipes/
- **Examples:** https://github.com/PrefectHQ/prefect/tree/main/examples

---

Solución de Problemas Comunes

Flow no se ejecuta en el schedule
1. Verificar que el worker esté corriendo: `docker compose ps`
2. Verificar que el deployment esté activo en la UI
3. Revisar los logs: `docker compose logs -f prefect-worker`

Tasks fallan con timeout

Incrementar el timeout en la configuración del task:

```
@task(timeout_seconds=600)  # 10 minutos
def task_lenta():
    pass
```

No se pueden guardar outputs

Verificar permisos de la carpeta outputs:

```
chmod -R 777 outputs/
```

Logs no aparecen en la UI

Verificar que estés usando `get_run_logger()`:

```
from prefect.logging import get_run_logger

@task
def mi_task():
    logger = get_run_logger()  # Usar esto
    logger.info("Mensaje")     # No print()
```

¡Ahora estás listo para crear workflows poderosos con Prefect!

Para deployment, consulta `deploy.md`.