



Seminario Técnico

# Automatización Inteligente

Python + Low-code Tools

## ÍNDICE DETALLADO (4 HORAS)

### 1 Introducción: La Realidad Híbrida

30 min

El muro del "No-Code" vs Python.  
Arquitectura JSON in/out.

### 2 Tour de Herramientas

45 min

Power Automate, n8n y Make  
(Python as Microservice).

### 3 Protocolo de Comunicación

45 min

Inputs (sys.argv/stdin) y Outputs  
(JSON serializado).

### 4 Caso A: Procesamiento Docs

1 hora

Parsing PDF, Regex y limpieza de  
datos.

### 5 Caso B: Enriquecimiento APIs

45 min

Lead Scoring, Requests y Lógica  
de Negocio.

### 6 Buenas Prácticas y Cierre

15 min

Dependencias, Seguridad  
(Secrets) y Q&A.

## MÓDULO 1

# De Scripts a Sistemas Cognitivos

La evolución tecnológica que hace posible la automatización híbrida hoy.

## Índice del Módulo

- 1 Historia: El Poder del Código (Python)
- 2 Revolución: La Promesa del No-Code
- 3 Conflicto: El Muro de la Complejidad
- 4 Futuro: LLMs y Convergencia

Historia

# 1991: Python y la Legibilidad

## "Executable Pseudocode"

Guido van Rossum diseñó Python con una filosofía clara: el código se lee mucho más de lo que se escribe.

Esto lo convirtió en el lenguaje ideal para **automatizar tareas aburridas** (Scripting).

- ✓ Sintaxis limpia (sin llaves {})
- ✓ "Batteries Included" (Librería estándar potente)

```
# Antes (Java/C++): Verboso y complejo public
class HelloWorld { public static void
main(String[] args) { System.out.println("Hello,
World"); } } # Python: Directo al grano
print("Hello, World") # Automatización en 3
líneas import shutil shutil.copy("report.pdf",
"/backup/folder")
```

Contexto

## La Revolución de las APIs (2000s)

El mundo dejó de ser "Software de Escritorio" y pasó a ser "SaaS en la Nube".

Para que Salesforce hablara con Gmail, nació el lenguaje universal: **JSON**.



💡 Aquí surge la oportunidad: "Necesitamos conectar estas APIs sin escribir código para cada conexión".

2011 - 2020

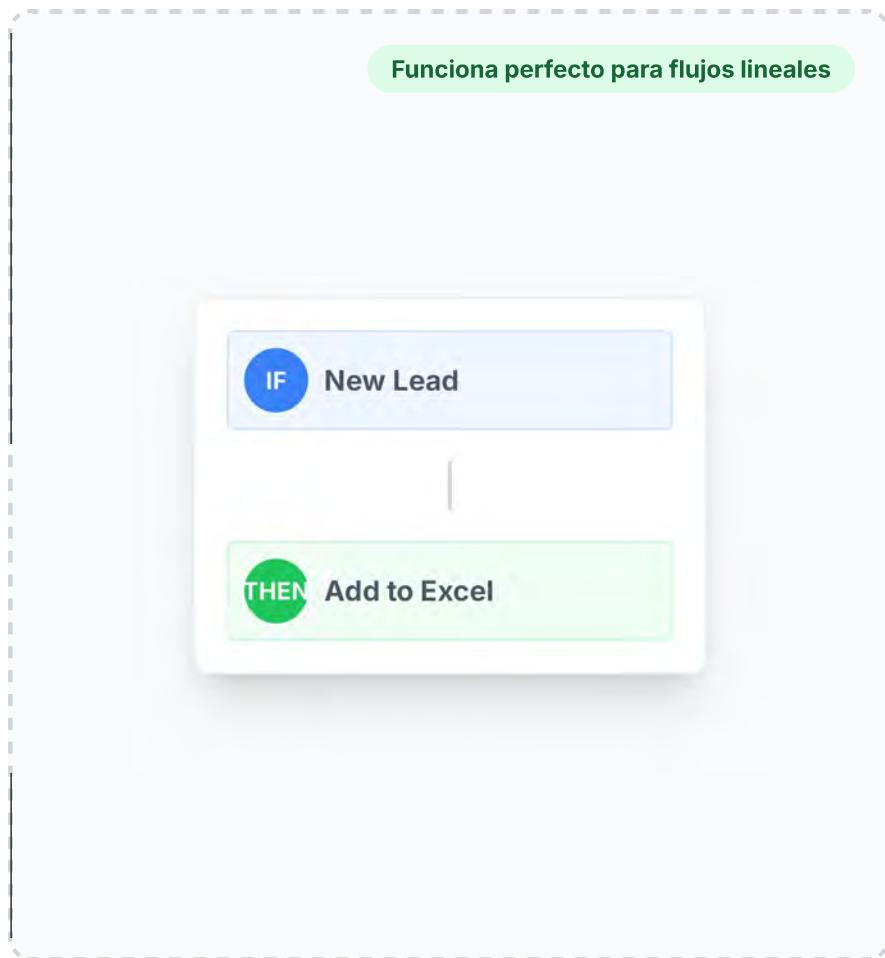
# La Promesa del No-Code

## "Democratizar la Automatización"

Herramientas como Zapier, IFTTT y luego Power Automate prometieron que cualquiera (Citizen Developer) podría integrar sistemas.

### La Propuesta de Valor:

- Trigger Visual:** "Cuando llegue un email..."
- Auth Gestionada:** Olvídate de OAuth2 tokens.
- Drag & Drop:** Mapeo de campos visual.



**El Problema**

# El Muro del "No-Code"

## La "Lasaña" de Bucles

Intentar replicar lógica compleja (ej. cruce de bases de datos) con cajas visuales resulta en flujos ilegibles y lentos.

### Timeouts

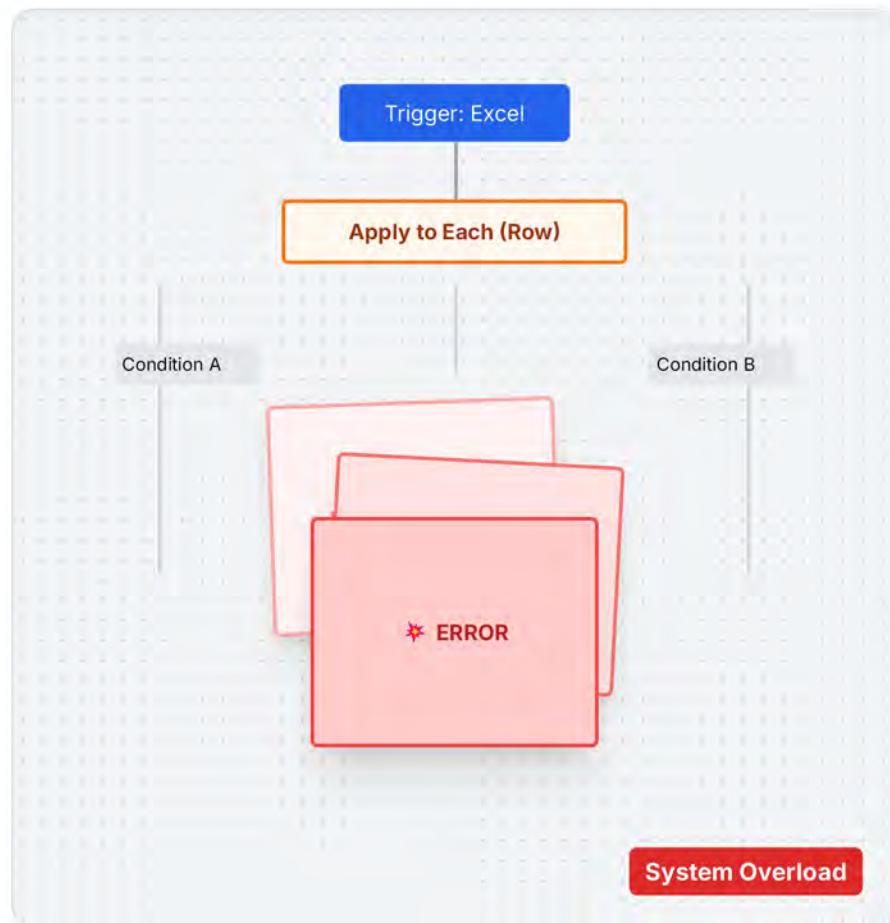
Power Automate falla tras 10 min de bucles.

### Depuración Imposible

¿Dónde falló la iteración 4,502?

### Coste de Operaciones

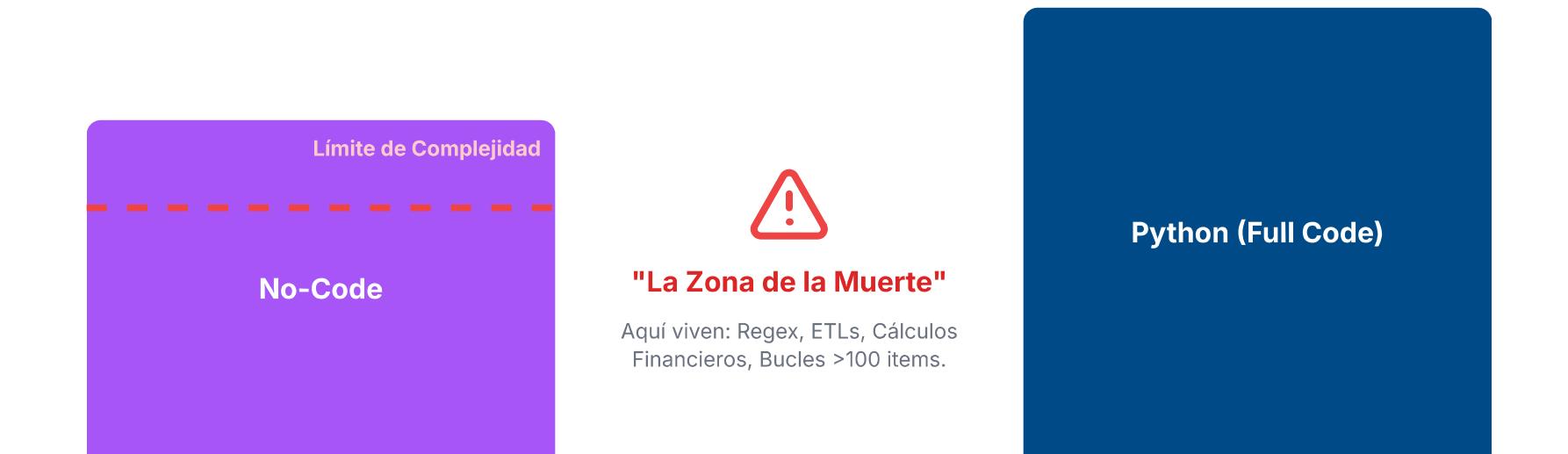
En herramientas como Make, pagas por cada "bolita".



Análisis

# La Brecha de Complejidad

El No-Code es excelente para mover datos (Transporte), pero terrible para transformar datos (Procesamiento).



2023 - Presente

# El Eslabón Perdido: IA Generativa

## La Barrera del Lenguaje ha caído

Antes, para saltar el "Muro del No-Code" necesitabas estudiar Python durante meses. Hoy, los LLMs (ChatGPT, Claude, Copilot) actúan como **Traductores Universales**.

### Nuevo Flujo de Trabajo:

1. Describes el problema en lenguaje natural (Español).
2. La IA genera el script de Python optimizado.
3. Pegas el script en tu herramienta Low-code.

Prompt

User: "Necesito un script que lea este JSON, filtre los usuarios mayores de 30 años y formatee sus nombres en mayúsculas."

```
import json

data = json.loads(input_json)
filtered = [
    {"name": u["name"].upper()}
    for u in data
    if u["age"] > 30
]
print(json.dumps(filtered))
```

**El Nuevo Estándar**

# Automatización Híbrida

## Lo mejor de ambos mundos



### No-Code (El Cuerpo)

Maneja las conexiones, autenticaciones y triggers.

**Orquestación**

+



### Python (El Cerebro)

Ejecuta la lógica compleja, regex, y transformación de datos.

**Procesamiento**

+



### LLMs (El Asistente)

Genera el código Python y estructura datos no estructurados.

**Generación**

Concepto

# La Analogía del Restaurante



## El Camarero (No-Code)

Se mueve rápido entre las mesas (Apps). Lleva y trae platos (Datos).

*Pero no le pidas que cocine un soufflé.*



## El Chef (Python)

No sale de la cocina (Caja Negra). Tiene las herramientas afiladas (Librerías).

*Transforma ingredientes crudos en el producto final.*

## El Error Común

- ⊗ Poner al camarero a cocinar (Hacer lógica compleja en Power Automate).

- ⊗ Poner al chef a servir mesas (Hacer un script de Python para cada integración simple).

**Resumen**

# Evolución del Flujo de Trabajo

Era	Herramienta Principal	Velocidad Desarrollo	Potencia
1. Scripting (2000s)	Python / Bash	Lenta	Alta
2. No-Code (2015s)	Zapier / Make	Muy Alta	Baja ( <b>Techo de cristal</b> )
3. Híbrida (Hoy)	Orquestador + Python Node	Alta	Ilimitada

**Conclusión:** Usamos No-Code para la estructura y Python para el detalle.



DEEP DIVE TÉCNICO

# Python al Rescate

Por qué 5 líneas de código valen más que 500 bloques visuales.

Vectorización, Librerías y Expresiones Regulares.

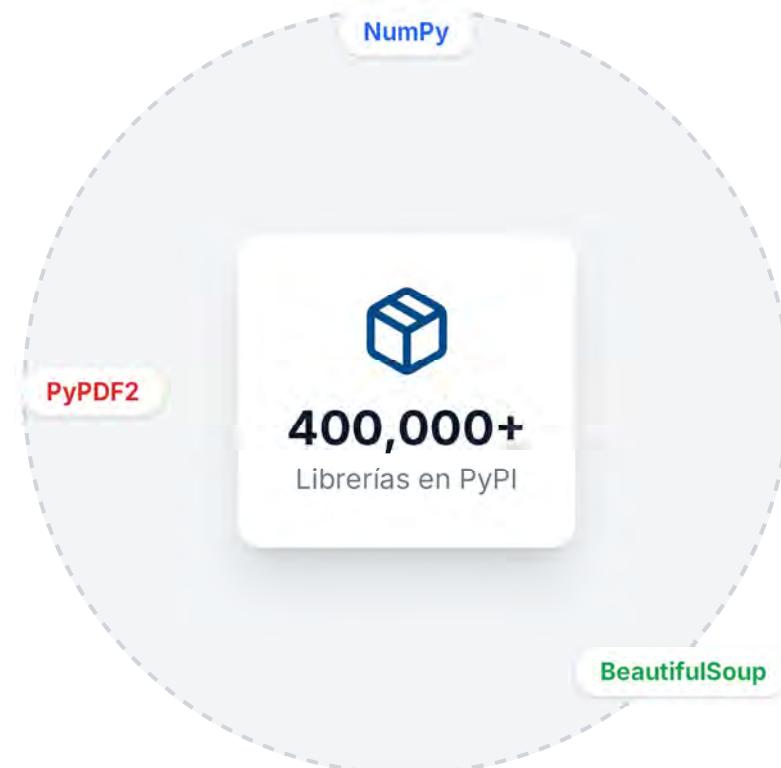
Concepto

# El Superpoder: PyPI (Ecosistema)

## No empiezas desde cero

Python no es rápido por sí mismo. Es rápido porque delega el trabajo pesado a librerías escritas en C y Fortran altamente optimizadas.

- Pandas** El "Excel con Esteroides". Manejo tabular de millones de filas.
- Requests** HTTP para humanos. Conecta con cualquier API sin dolor.
- OpenPyXL** Lectura/Escritura nativa de archivos .xlsx sin abrir Excel.



Rendimiento

# El Secreto: Vectorización

En Low-code (y programación tradicional), procesas fila por fila.

En Pandas, procesas **toda la columna a la vez**.

## Low-Code / Bucles

1 Procesando...

2 Esperando...

3 Esperando...

Tiempo: 10 min

## Pandas (SIMD)

1 2 3 4 ...

La CPU aplica la operación a todo el bloque de memoria simultáneamente.

Tiempo: 0.2 seg

Caso 1

# Limpieza de Datos (Data Cleaning)

## El Problema:

Los Excel's nunca vienen perfectos. Celdas vacías, fechas como texto, espacios en blanco...

En Low-code, necesitarías condicionales anidados:

*"Si la celda está vacía, pon 0, si no, mira si es texto..."*

clean\_data.py

```
import pandas as pd df =  
pd.read_excel("sucio.xlsx") # 1. Eliminar filas  
vacías completamente df = df.dropna(how='all') #  
2. Rellenar Nulos con 0 en columna 'Ventas'  
df['Ventas'] = df['Ventas'].fillna(0) # 3.  
Quitar espacios en blanco de nombres  
df['Cliente'] =  
df['Cliente'].str.strip().str.upper() # 4.  
Convertir fecha (maneja errores auto)  
df['Fecha'] = pd.to_datetime(df['Fecha'],  
errors='coerce') print(df.to_json())
```

Caso 2

# El Cirujano de Texto: Regex

## Input Text (PDF Sucio)

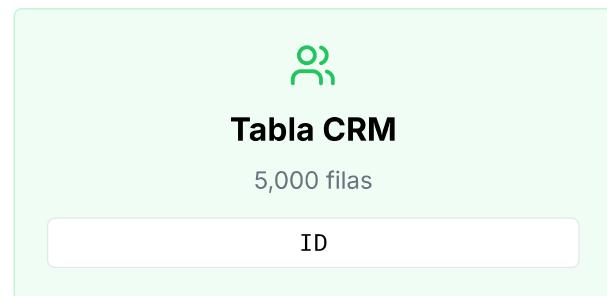
```
"Factura Num: 9928. Contacto: admin@tesla.com.  
Teléfono soporte: +34 600-111-222.  
Pagar antes del 2025/12/01."
```

Extraer patrones complejos en herramientas visuales requiere dividir cadenas (`split`), buscar índices (`indexOf`) y es muy frágil.

```
import re  
text = "Factura Num: ..." # Extraer  
# TODOS los emails  
emails = re.findall(r'[\w\.-]+\@\[\w\.-\]+\.', text) # →  
['admin@tesla.com'] # Extraer teléfonos (formato  
flexible)  
phones = re.findall(r'\+?\d[\d -]{8,12}\d', text) # → ['+34 600-111-222'] #  
# Extraer Fechas  
dates = re.findall(r'\d{4}/\d{2}/\d{2}', text) # → ['2025/12/01']
```

Caso 3

# El "Merge" (VLOOKUP Masivo)



## La Línea Mágica

```
df_final = df_ventas.merge( df_crm, left_on='ID_Cliente', right_on='ID', how='left' )
```

✓ Maneja duplicados

✓ Maneja tipos de datos diferentes

✓ Memoria eficiente

**Estabilidad**

# Robustez y Manejo de Errores



## Low-Code Crash

Si una fila falla, a menudo **todo el flujo se detiene**.  
Configurar "Run after failure" en cada nodo es tedioso.

## Python Try / Except

Control granular. Si falla un dato, lo registramos y seguimos.

```
processed_rows = [] errors = [] for row in data:  
    try: # Lógica arriesgada res = 100 /  
        row['valor'] processed_rows.append(res) except  
        Exception as e: # No detenemos el robot, solo  
        anotamos errors.append({"row": row, "error":  
        str(e)}) # Al final, devolvemos éxito Y reporte  
        de errores print(json.dumps({"ok":  
        processed_rows, "err": errors}))
```

**Conectividad**

# APIs Complejas

El nodo "HTTP Request" de las herramientas visuales es básico.  
¿Qué pasa cuando la API requiere lógica antes de responder?



## Paginación

"Mientras 'next\_page' exista, sigue pidiendo". Hacer un bucle 'while' en visual es doloroso.



## Auth Dinámica

Firmas HMAC, encriptación de payloads o tokens que rotan cada 5 minutos.



## Rate Limits

Lógica de "Backoff exponencial" (esperar progresivamente si la API da error 429).

Negocio

# Escalabilidad y Costes

Volumen de Datos	Solución Low-Code	Solución Python
10 filas	Perfecto (Rápido)	Overkill (Excesivo)
10,000 filas	Lento (Minutos)	Instantáneo (Segundos)
1,000,000 filas	Timeout / Crash	Robusto (Chunks)
Coste (SaaS)	\$\$\$ (1 millón de operaciones)	\$ (1 operación de ejecución)



# Conclusión Técnica

## Cuándo usar Python:

- Transformación de Arrays/Listas grandes.
- Limpieza de texto compleja (Regex).
- Cálculos matemáticos/estadísticos.
- Manejo de archivos binarios (PDF, Excel, Zip).

## Cuándo NO usar Python:

- Enviar un email simple.
- Aprobar un documento (Interacción humana).
- Triggers nativos (Cuando llega un Forms).

Siguiente: Arquitectura del Flujo Híbrido ↓

MÓDULO 1.3

# Arquitectura del Flujo Híbrido

Anatomía detallada de un sistema de automatización robusto,  
escalable y mantenable.

 Low-Code

 Python

 LLMs

Visión Global

# El Pipeline de Datos



Paso 1

# El Trigger: Por qué Low-code gana aquí

Escribir "listeners" en Python puro es costoso de mantener (requiere servidor 24/7). Las herramientas Low-code abstraen esta complejidad en 3 tipos:

## ⚡ Webhook (Instantáneo)

La app externa "empuja" los datos. (Ej. Typeform envía JSON al terminar).

## ⌚ Polling (Consultas)

El robot pregunta cada 5 min: "¿Hay algo nuevo en esta carpeta?".

## 🕒 Schedule (Cron)

Ejecución programada (Ej. "Todos los viernes a las 17:00").



## Python Serverless

Nuestro script de Python está "dormido". El Low-code actúa como el despertador.

**Ahorro masivo de recursos.**

**Paso 2**

# Payload: El Contrato JSON

Lo que envía Low-code

## MAL PAYLOAD (TEXTO PLANO)

"Hola, soy Juan, mi ID es 123 y quiero..."

Obliga a Python a parsear lenguaje natural (frágil).

## BUEN PAYLOAD (ESTRUCTURADO)

```
{ "meta": { "source": "typeform", "timestamp": "2023-10-01T10:00:00Z" }, "data": { "user_id": 123, "raw_text": "Hola, soy Juan...", "files": ["url_pdf_1", "url_pdf_2"] } }
```

Datos limpios listos para `json.loads()`.



## Schema Validation

Python debe validar la entrada antes de procesar.

```
if 'data' not in payload: raise ValueError
```

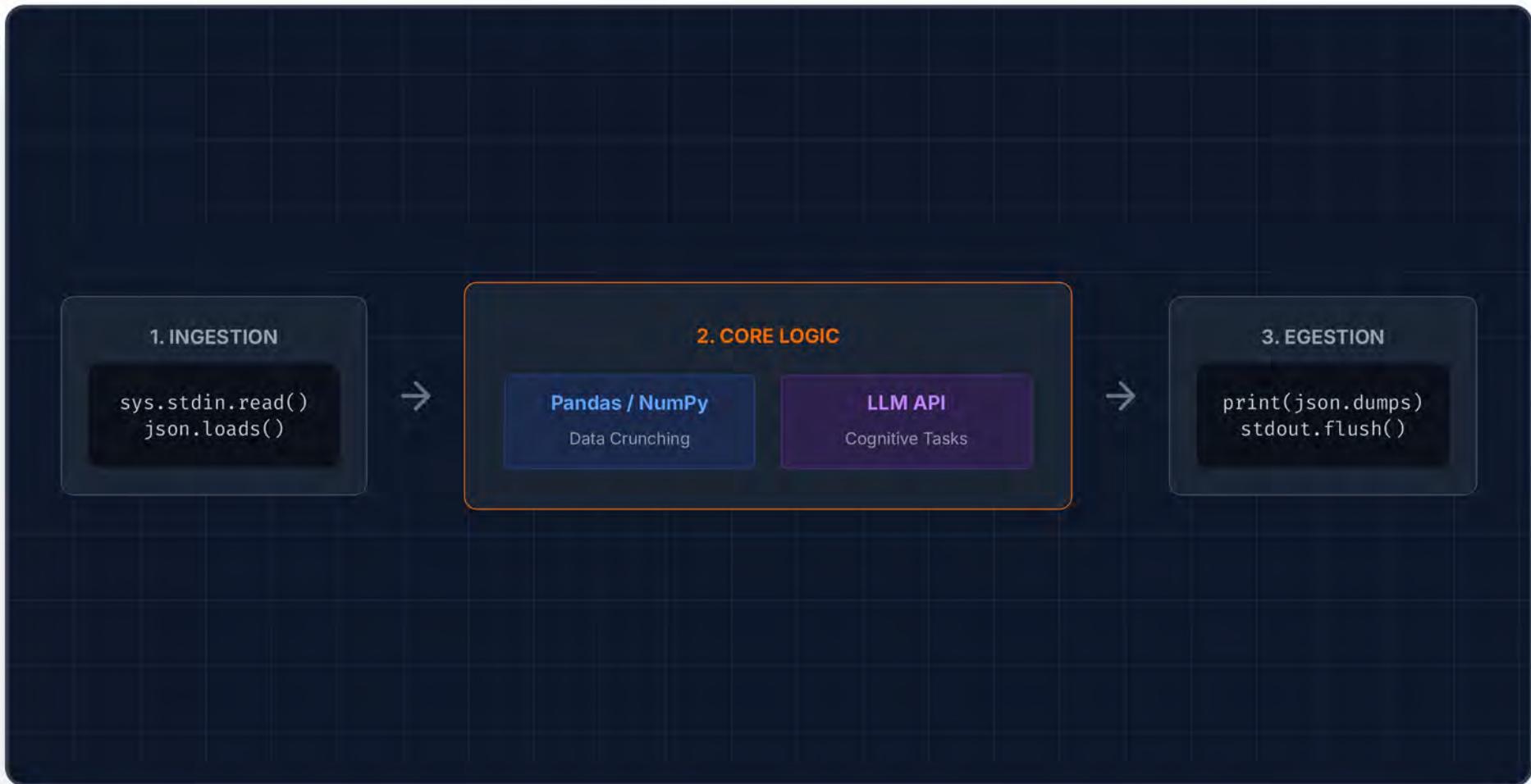


## Minimalismo

No envíes todo el objeto de SharePoint. **Envía solo lo que Python necesita.** Reduce latencia y costes.

**Paso 3**

# Dentro de la Caja Negra



IA

# El Rol del LLM en la Arquitectura

## No es magia, es una función más.

Dentro del script de Python, el LLM (OpenAI, Claude) actúa como una subrutina para tareas donde el código determinista falla.



### Pre-procesamiento (Unstructured → Structured)

Convertir un email libre en JSON: {"intent": "complaint", "urgency": "high"}.



### Post-procesamiento (Generación)

Redactar una respuesta empática basada en los datos calculados por Pandas.

```
def ask_llm(text): prompt = f"Extrae entidades de: {text}" response = openai.ChatCompletion.create( model="gpt-4-turbo", messages=[{"role":"user", "content":prompt}]) return json.loads(response.choices[0].message) # Flujo Híbrido raw_data = sys.stdin.read() structured = ask_llm(raw_data) # IA estructura final_calc = pandas_logic(structured) # Python calcula print(json.dumps(final_calc))
```

**Persistencia**

# ¿El script tiene memoria?

**Opción A: Stateless**

## Amnesia Total

El script nace, corre y muere. No sabe qué pasó en la ejecución anterior.

Ideal para transformaciones puras.

**Opción B: Stateful**

## Memoria Externa

Python conecta a una DB (SQL/Redis) para leer el contexto.

"¿Ya procesé este ID ayer?"

Necesario para deduplicación incremental.

**Seguridad**

# Gestión de Secretos (Vault)

## ⚠️ Anti-Patrón

Nunca escribir API Keys dentro del código Python (`api\_key = "sk-..."`). Si compartes el script, expones la llave.

## 🔒 Best Practice

Usar Variables de Entorno injectadas por la herramienta Low-code.

### Environment Injection

```
# En Power Automate / n8n (Config)
ENV_VAR_OPENAI_KEY = *****

# En Python Script
import os

key = os.environ.get('ENV_VAR_OPENAI_KEY')
if not key:
    raise ValueError("Missing Key")
```

Resiliencia

# Arquitectura de Errores



## Happy Path (STDOUT)

Todo salió bien. Python devuelve el JSON resultado.

```
{ "status": "success", "data": [ ... ] }
```



## Crash Path (STDERR)

Excepción no controlada o error crítico.

```
Traceback (most recent call last):
  File "script.py", line 10, in 
    ZeroDivisionError: division by zero
```

Low-code detecta "Exit Code 1" y lanza alerta.

Paso 4

## La Acción Final: ¿Por qué volver?

### No reinventes la rueda

Podríamos enviar el email desde Python (`smtplib`), pero configurar OAuth2 de Outlook/Gmail en código es una pesadilla de mantenimiento.

#### Filosofía Híbrida:

"Usa Python para lo difícil (datos) y Low-code para lo conectado (integraciones)."



#### Human-in-the-Loop

Approval Flows (Teams/Slack) son nativos en Low-code.



#### Auth Gestionada

Conectores pre-fabricados (Salesforce, SAP, Jira).



#### Auditoría Visual

Es más fácil ver un check verde en el historial de Power Automate.



MÓDULO 3

# Tour de Herramientas

Dónde escribir el código. Dónde ejecutar el código.

RPA vs iPaaS vs Self-Hosted.

Panorama

# El Espectro de Ejecución

No todas las herramientas ejecutan Python igual. La elección depende de **dónde viven tus datos**.

## Desktop (Local)



El script corre en TU máquina.

- ✓ Acceso a C:/Archivos
- ✓ Drivers Corporativos
- ✓ Coste: Licencia Windows

## Server (Híbrido)



El script corre en un contenedor Docker.

- ✓ Control total del entorno
- ✓ Ejecución rápida
- ✓ Coste: Servidor (5\$/mes)

## Cloud (SaaS)



El script corre en AWS/Azure (Microservicio).

- ✓ Escalabilidad infinita
- ✓ Cero mantenimiento
- ✓ Coste: Por operación



MÓDULO 2.1 • DEEP DIVE

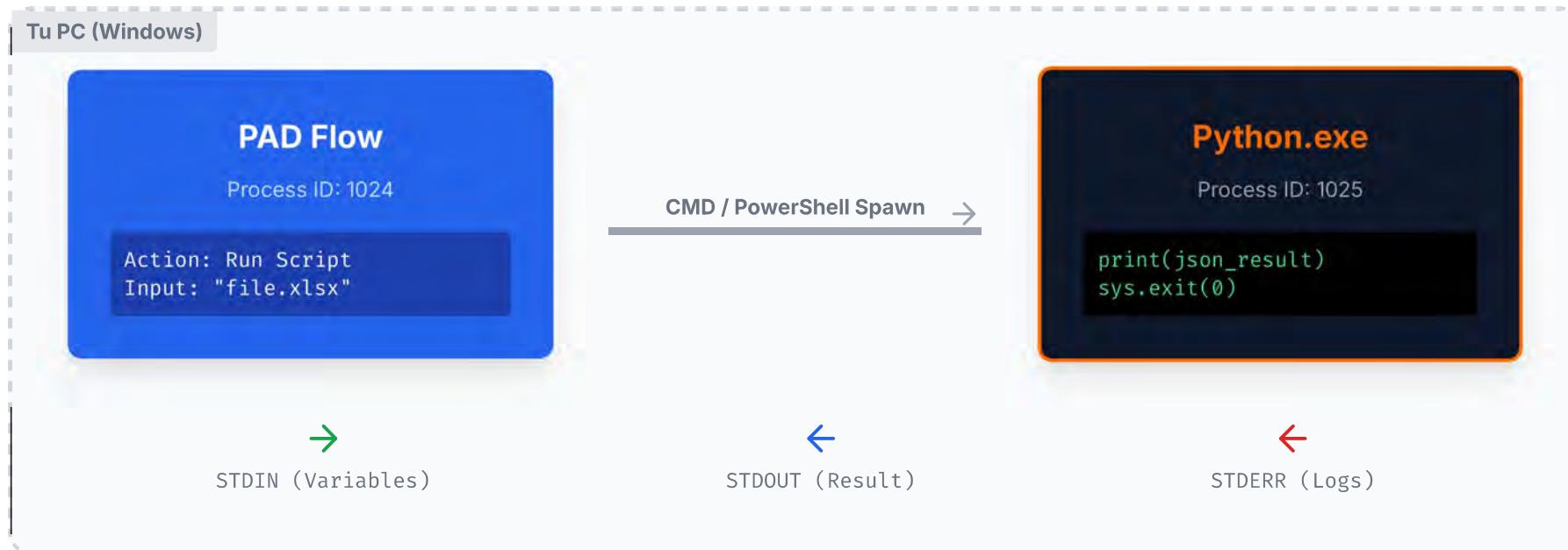
# Power Automate Desktop

Cómo domar la integración local de Python en entornos Windows.

**Configuración, Entornos Virtuales y Hacks de Registro.**

Arquitectura

# ¿Qué ocurre realmente?



**La Herramienta**

# La Acción: "Run Python Script"



## Modo 1: Código Directo

Escribes el código dentro de la cajita de texto de PAD.

Bueno para scripts de < 10 líneas.

## Modo 2: Importar Archivo (Recomendado)

En la caja escribes solo:

```
import sys
sys.path.append(r'C:\Scripts')
import mi_script_complejo
```

Permite usar Git y editores reales (VS Code).

**Troubleshooting**

# El Problema #1: ¿Qué Python soy?

## ! El Síntoma

```
ImportError: No module named 'pandas'
```

Tú instalaste pandas (`pip install pandas`), pero PAD está usando una versión de Python diferente a la que tú usas en la terminal.

## La Solución (Configuración PAD)

### Paso 1: Averigua tu ruta real

```
where python (en CMD)
```

### Paso 2: Configura PAD

En la configuración de flujo, o forzando el registro de Windows.

*PAD busca en el Registro de Windows la clave 'HKLM\Software\Python\PythonCore'.*

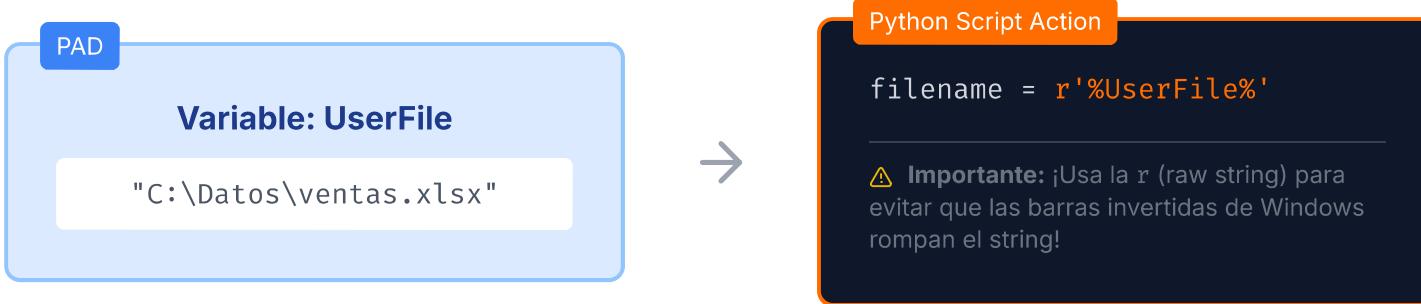
### El Truco Pro (Shebang Virtual)

No dependas de la detección automática. Llama al ejecutable absoluto en tu script wrapper.

Datos

# Inputs: De PAD a Python

Power Automate usa la sintaxis `'%NombreVariable%'`. Cuando el script corre, PAD hace un "Find & Replace" literal antes de ejecutar.



**Cuidado con los tipos:** PAD siempre inyecta texto. Si pasas un número `'%Edad%'`, en Python recibirás "25" (string). Recuerda hacer `int()`.

Datos

# Outputs: De Python a PAD

## El Mecanismo STDOUT

PAD captura **todo** lo que tu script imprime en la consola (`'print()'`) y lo mete en una variable llamada `'%PythonScriptOutput%'`.

## El Peligro de los Logs

Si haces esto:

```
print("Iniciando proceso ... ")
print(json_resultado)
```

PAD recibirá basura mezclada. Solo debes imprimir el resultado final.

Correct Pattern

```
import json

data = procesar_datos()

# No imprimas logs intermedios a stdout!
# Úsalos a un archivo o stderr si es necesario.

print(json.dumps(data))
```

**Avanzado**

# Encoding: "Ñ" y Tildes

## UnicodeDecodeError

Windows usa por defecto `cp1252` (Western Europe). Python 3 usa `utf-8`.

Cuando PAD lee el output de Python, a menudo corrompe los caracteres especiales (ej. "Camión" → "CamiÃ³n").

### 🔧 El Fix Obligatorio

Añade esto AL PRINCIPIO de todos tus scripts para PAD:

```
import sys import io # Forzar salida estándar a UTF-8
sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding='utf-8')
```

**Best Practice**

# Entornos Virtuales (VENV)

## Cómo invocar VENV desde PAD

### MAL MÉTODO (GLOBAL)

```
python script.py
```

### BUEN MÉTODO (VENV ABSOLUTE PATH)

```
C:\Bots\Finanzas\venv\Scripts\python.exe  
script.py
```

En la acción de PAD, no uses solo "python".  
Usa la ruta completa al ejecutable dentro de la carpeta 'venv'.

## No ensucies tu sistema

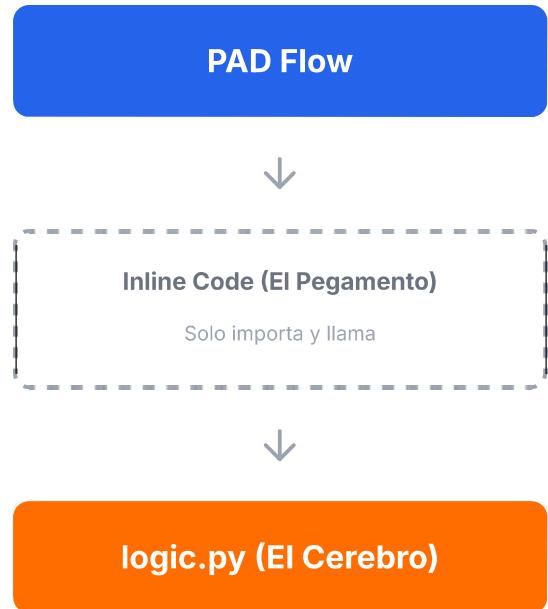
Si tienes varios bots, cada uno podría necesitar versiones diferentes de pandas. Usar el Python global es una receta para el desastre.

 Bot Finanzas: pandas 1.3

 Bot Marketing: pandas 2.0

Arquitectura

# Patrón "Wrapper"



## ¿Por qué separar?

1. Puedes testear `logic.py` fuera de PAD.
2. Puedes usar Git para versionar `logic.py`.
3. PAD es un editor de código terrible.

```
import sys
sys.path.append(r'C:\MisBots\Lib')
import logic_processor # Inyección de dependencias (Variables de PAD)
file = r'%FilePath%'
try:
    result = logic_processor.run(file)
    print(result)
except Exception as e:
    print(f"ERROR: {e}", file=sys.stderr)
```

**Cierre**

# Resumen: El Camino del Éxito

## 👍 Do's (Hacer)

- Usar rutas absolutas a `python.exe` (venv).
- Imprimir SOLO JSON al final (`print(json.dumps)`).
- Forzar `utf-8` en stdout.
- Usar `raw strings` (r'') para rutas de Windows.

## 👎 Don'ts (Evitar)

- Escribir lógica de negocio compleja directo en PAD.
- Imprimir mensajes de depuración ("Hola", "Paso 1") en producción.
- Asumir que la librería instalada globalmente estará disponible.



MÓDULO 2.2 • DEEP DIVE

# n8n: The Developer's Tool

Automatización Fair-code para ingenieros.  
**Docker, Estructuras de Datos y Ejecución Híbrida.**

**Concepto**

# Anatomía de los Datos

## No es un JSON plano

El error #1 de los desarrolladores en n8n es asumir que reciben un diccionario simple.

n8n procesa un **Array de Objetos**, donde cada objeto tiene una estructura fija: json para datos y binary para archivos.

⚠ Si intentas acceder a `input['email']` fallará.  
Debes usar `input['json']['email']`.

**n8n Data Structure**

```
[ { "json": { "id": 1, "name": "Juan", "role": "Admin" }, "binary": { "avatar": { ... } } }, { "json": { "id": 2, "name": "Ana" } } ]
```

Interfaz

# El Nodo "Code"

<> Run Custom Python

Mode: Run Once for All Items

Python 3

```
for item in _input.all():
    item.json["new_value"] = item.json["price"] * 1.21

return _input.all()
```

INPUT DATA

```
[ { "json": { "price": 100 } } ]
```

OUTPUT DATA

```
[ { "json": { "price": 100,
  "new_value": 121 } } ]
```

Lógica

# Estrategia de Ejecución

## ▷ Run for Each Item

El script se ejecuta X veces (una por cada fila de entrada).  
Es como un bucle map( ) implícito.

```
# Python ve solo 1 objeto a la vez  
item.json['tax'] = item.json['cost'] * 0.2
```

## ▷ Run Once for All Items

El script se ejecuta 1 sola vez recibiendo todo el array.  
Es necesario para agregaciones, sumas totales o  
comparaciones entre filas.

```
# Python ve todo el array  
total = sum(i.json['cost'] for i in _input.all())
```

**Seguridad**

# El Dilema: Sandbox vs Realidad

## Pyodide (Default)

Seguro

Python compilado a WebAssembly. Corre aislado en memoria.

**Limitación:** Solo soporta librerías pure-Python (micropip). No funciona 'pandas' complejo ni acceso a disco.



## Native (Docker)

Potente

Python real corriendo en el contenedor del servidor.

**Poder:** Acceso a todo PyPI, Filesystem, y Hardware.

**Riesgo:** Si rompes el script, puedes colgar el nodo de n8n.



**DevOps**

# Habilitar Python Nativo

Para usar librerías potentes como pandas o scikit-learn, necesitamos construir una imagen de Docker personalizada de n8n.

## ⚠ Configuración Requerida

En tu archivo .env de Docker Compose:

```
N8N_RUNNERS_ENABLED=true
```

**Dockerfile**

```
FROM n8nio/n8n:latest

# Cambiar a usuario root para instalar
USER root

# Instalar Python y Pip
RUN apk add --update python3 py3-pip

# Instalar Librerias Cientificas
RUN pip3 install pandas numpy requests beautifulsoup4

# Volver al usuario n8n (Seguridad)
USER node
```

**Código**

# Usando Librerías Externas

**1**

## Habilitar Imports

Por seguridad, n8n bloquea los imports. Debes permitirlos en la variable de entorno:

```
NODE_FUNCTION_ALLOW_EXTERNAL=pandas,requests,numpy
```

**2**

## Usar en el Nodo Code

```
import pandas as pd import requests # Ahora puedes usar pandas con los datos de entrada df = pd.DataFrame(_input.all()) result = df.describe().to_dict() return [{"json": result}]
```

**Avanzado**

# Memoria entre Ejecuciones

## El Problema de la Amnesia

Normalmente, cada ejecución del workflow es independiente. ¿Cómo sabemos si ya procesamos el último tweet o el último email?

### Solución: `getWorkflowStaticData`

Un pequeño almacén de datos persistente exclusivo de ese workflow.

```
from n8n.core.node import getWorkflowStaticData
# Acceder a la memoria static_data =
getWorkflowStaticData('global') last_id =
static_data.get('last_id', 0) # Filtrar datos
nuevos new_items = [i for i in _input.all() if
i.json['id'] > last_id] # Actualizar memoria si
hay nuevos if new_items: static_data['last_id'] =
new_items[-1].json['id'] return new_items
```

# Gestión de Fallos



## Error Workflow

### Error Trigger Node

Se activa automáticamente cuando el flujo principal muere.

```
{ "msg": "404", "node": "HTTP Request" }
```

### Continue on Fail

Configuración del nodo. Si falla, el flujo sigue pero marca el item como error.

### Retry Strategy

Reintentar 3 veces con espera de 5 segundos antes de rendirse.



## Mini ETL en n8n

Flujo completo: Webhook → Python (Limpieza) → Google Sheets



[⬇ Descargar Blueprint JSON](#)



MÓDULO 2.3 • DEEP DIVE

# (Integromat)

El arte de conectar lo inconectable.  
IFTTT, Zapier, Webhooks y Estrategias de Coste.

Filosofía

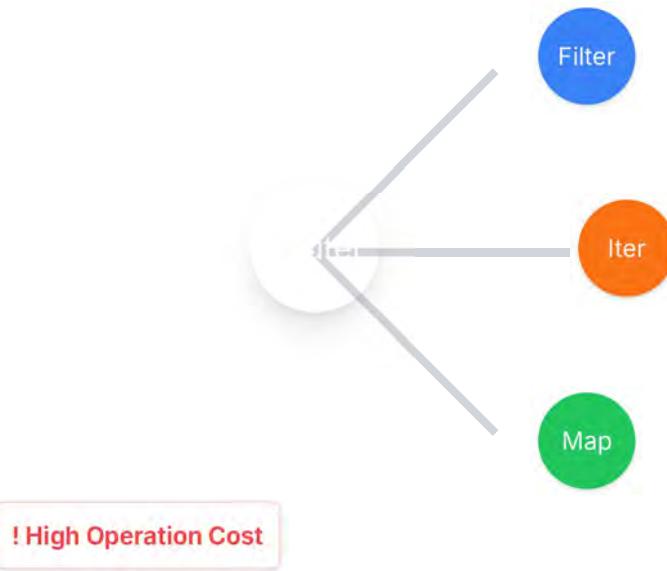
# El Jardín Amurallado Visual

## "No-Code First"

Make odia el código. Su misión es abstraer la complejidad en burbujas visuales. Si quieres transformar datos, te obligan a usar su propio lenguaje de fórmulas (similar a Excel).

### El Dolor del Desarrollador:

Hacer un bucle 'for' anidado con condiciones en Make requiere múltiples módulos, aumenta la complejidad visual y dispara el coste.



Arquitectura

# Externalizando el Cerebro

En lugar de luchar contra Make, usamos Make para lo que es bueno (Conectores) y delegamos la lógica a la nube.



**Tecnología**

# ¿Qué es una Cloud Function (FaaS)?

Es un trozo de código que vive en la nube y solo "despierta" cuando lo llamas.



## Pago por uso real

Si nadie lo llama, pagas \$0. El "Free Tier" suele incluir 1-2 millones de llamadas al mes.



## Auto-Escalado

Si Make envía 1,000 peticiones a la vez, la nube crea 1,000 copias de tu script instantáneamente.

## PROVEEDORES PRINCIPALES

### AWS Lambda

Estándar industrial. Capa gratuita de por vida.

### Azure Funcs

Integración perfecta si usas Power Automate.

### Google Cloud

Muy fácil de desplegar (Source-to-function).

**Código**

# El Esqueleto del Microservicio

main.py (Google Cloud Functions style)

```
import functions_framework import pandas as pd import json
@functions_framework.http def process_data(request): try: # 1. Leer
    JSON de Make request_json = request.get_json(silent=True) if not
    request_json: return 'No JSON', 400 # 2. Lógica Pandas (Lo que Make
    no puede hacer) df = pd.DataFrame(request_json['data']) result =
    df.groupby('category').sum().to_dict() # 3. Devolver JSON limpio
    return json.dumps(result), 200, {'Content-Type': 'application/
    json'} except Exception as e: return json.dumps({"error": str(e)}),
    500
```

## Request Object

El objeto `request` contiene todo lo que envía Make (Body, Headers, Params).

## Return Tuple

Devolvemos (Cuerpo, Código HTTP, Headers). Make leerá esto automáticamente.

# El Nodo "HTTP Request"

HTTP Make A Request

URL  
`https://us-central1-project.cloudfunctions.net/proc...`

METHOD  
`POST`

BODY TYPE  
`Raw`

CONTENT TYPE  
`JSON (application/json)`

REQUEST CONTENT

```
{  
  "data": {{json_array_from_previous_module}}  
}
```

## Parse Response

Asegúrate de marcar la casilla "**Parse response**". Esto convierte el JSON que devuelve Python en variables usables en Make automáticamente.

## Timeout Settings

El nodo HTTP de Make tiene un timeout por defecto (aprox 40-60s). Si tu script tarda más, necesitas un patrón asíncrono.

**Seguridad**

# Protegiendo tu Microservicio

Tu Cloud Function es pública en internet. Cualquiera con la URL podría ejecutarla y gastar tu dinero si no la proteges.

Make (Header)

X-API-KEY: secret123



Python (Validation)

```
secret = req.headers.get('X-  
API-KEY')  
if secret !=  
os.environ['MY_KEY']:  
    return 401
```

Usa **Variables de Entorno** en la configuración de la Cloud Function para guardar el secreto. Nunca lo hardcodees.

**Avanzado**

# Scripts que tardan mucho (>60s)

## El problema del Timeout

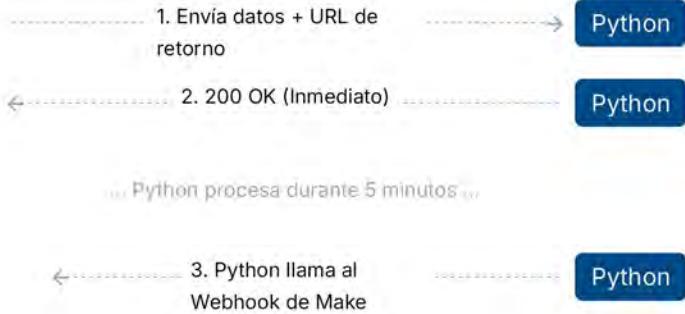
Make cerrará la conexión si Python no responde rápido.

Para procesos largos (ej. Video Rendering, Scraping masivo), cambiamos el patrón.

**Sync** Make espera respuesta... (Timeout)

**Async** Python dice "Recibido, te llamo luego".

## Patrón Callback (Webhook de Respuesta)



ROI

# La Economía de la Operación

## Coste en Make (Nativo)

Si intentas procesar 10,000 filas con módulos de Make (Iterador + Aggregator + JSON Parse):

**~30,000 Ops**

10k iteraciones \* 3 módulos = 30k operaciones.  
Esto puede consumir tu plan mensual en un día.

## Coste Híbrido (Python)

Envías las 10,000 filas en un solo JSON a Python. Python procesa y devuelve el resultado.

**2 Ops**

1 llamada HTTP + 1 respuesta.  
Coste de AWS Lambda para 1 seg de ejecución: **\$0.000016**.

**Conclusión: Usar Python externo ahorra un 99% de "Operaciones" en tareas de datos masivos.**



# Cuándo usar esta arquitectura

## ESCENARIOS IDEALES

- ETLs complejos (Transformar JSON anidados).
- Uso de librerías específicas (Pandas, Numpy, CV2).
- Ahorro de costes en volúmenes altos.

## NO VALE LA PENA SI...

- Es una lógica condicional simple (If/Else).
- Solo mueves datos de A a B sin cambios.
- No tienes conocimientos para mantener la Cloud Function.

**Decisión**

# Matriz de Selección

Característica	Power Automate (PAD)	n8n	Make
Ejecución Python	Local (Nativo)	Nativo (Docker/Pyodide)	Externa (Vía API)
Curva Aprendizaje	Media (Microsoft UI)	Alta (Técnica)	<b>Baja (Visual)</b>
Coste	Incluido en O365	<b>Gratis (Self-hosted)</b>	\$\$ (Por operación)
Mejor uso	Excel Local / Legacy Apps	Data Heavy / ETLs	SaaS Glue (Marketing)



## Fin del Tour

No te cases con una herramienta.

Usa **Make** para prototipar rápido, **n8n** para procesar datos gratis, y  
**PAD** para mover el ratón en Windows.

SIGUIENTE MÓDULO

**El Protocolo JSON**



MÓDULO 3 • MASTERCLASS

# El Protocolo de Comunicación

Cómo Low-code y Python hablan el mismo idioma sin corromper los datos.

**STDIN, STDOUT, JSON y Exit Codes.**

Concepto

# La Torre de Babel

Python es un entorno aislado. Power Automate/n8n son otros.  
No comparten memoria RAM. No comparten variables.

**Low-code**

"Tengo un objeto Customer"



¿Cómo te lo paso?

**Python**

"Yo solo entiendo Strings"

Estándar

# JSON: El Lenguaje Universal

## JavaScript Object Notation

Es el único formato que todas las herramientas modernas entienden nativamente.

- ✓ Legible por humanos.
- ✓ Compacto (vs XML).
- ✓ Tipado básico (String, Number, Bool).

```
{  
  "string": "Hola",  
  "number": 42,  
  "float": 3.14,  
  "boolean": true,  
  "array": [1, 2, 3],  
  "object": { "key": "val" },  
  "null": null  
}
```

**Reglas**

# Reglas de Oro del JSON

## 1. Comillas Dobles Estrictas

{ "key": "value" } ✓      { 'key': 'value' } ✗ (Esto es Python dict, no JSON)

## 2. Sin Comas Finales (Trailing Commas)

[1, 2, 3] ✓      [1, 2, 3,] ✗ (Rompe el parser)

## 3. Tipos de Datos Limitados

No existen fechas ('Date'), funciones ni comentarios en JSON estándar.

**Estructura**

# Estructuras: {} vs []



## Objeto (Diccionario)

Pares Clave-Valor desordenados.

Uso: Representar una entidad (Un Cliente).



## Array (Lista)

Colección ordenada de valores.

Uso: Representar una lista (Muchos Clientes).

Complejidad

## Nesting: El Poder Real

JSON permite meter objetos dentro de listas, dentro de objetos. Esto modela la realidad perfectamente.

**Ejemplo:** Un Pedido (Objeto) tiene una lista de Items (Array), donde cada Item tiene detalles (Objeto).

```
{ "id": 101, "items": [ { "name": "Laptop", "price": 1000 }, { "name": "Mouse", "price": 20 } ] }
```



## Parte 2: Inputs

Cómo meter datos desde la herramienta visual hacia el script de Python.

**Método A**

## Argumentos de Línea de Comandos

```
python script.py "Juan" 35 "Madrid"
```

Es como llamar a una función. Pasas los valores uno detrás de otro separados por espacios.

```
import sys
nombre = sys.argv[1] # "Juan"
edad = sys.argv[2] # "35"
```

**Limitaciones**

# Por qué evitar sys.argv

## 1. Límite de Caracteres

Windows CMD tiene un límite de ~8191 caracteres. Si pasas un JSON grande o un Base64, se cortará.

## 2. Problemas de Escapado

Si tu texto contiene comillas " o espacios, puede romper el comando si no se escapa perfectamente.

## 3. Inseguro

Los argumentos son visibles en el administrador de tareas (ps/top). No pases contraseñas por aquí.

**Método B**

# Standard Input (STDIN)

## La Tubería (Pipe)

En lugar de pasar datos en el comando de arranque, los "inyectamos" una vez el proceso ha iniciado.

Es como copiar y pegar un libro entero en la consola mientras el programa corre.

- Sin límite de tamaño.
- Seguro (no visible en logs de sistema).
- Perfecto para JSONs complejos.

```
import sys
import json

# Leer todo el buffer de entrada
raw_input = sys.stdin.read()

data = json.loads(raw_input)
print(f"Recibido: {len(data)} items")
```

**Práctica**

# STDIN en n8n

**Execute Command Node****COMMAND**

```
python3 script.py
```

**✓ INPUT DATA (STDIN)**

```
{{ JSON.stringify($json) }}
```

Aquí mapeas el JSON que quieras enviar.



## Parte 3: Python Processing

Cómo convertir ese texto JSON en objetos nativos de Python  
(Diccionarios y Listas).

**Función Clave**

# json.loads()

## Load String

Toma una cadena de texto (String) y la convierte en estructuras de Python.

```
String "true" → Python True (bool)
String "null" → Python None (NoneType)
String "{}" → Python {} (dict)
String "[]" → Python [] (list)
```

```
import json json_str = '{"nombre": "Ana",
"edad": 25}' # Deserializar data =
json.loads(json_str) # Ahora es un diccionario
print(data['nombre']) # Output: Ana
```

**Defensa**

# JSONDecodeError

## El Peligro

Si el Low-code envía un JSON mal formado (ej. falta una comilla, o llega vacío), `json.loads()` lanzará una excepción y tu script morirá.

```
try: data = json.loads(raw_input) except  
    json.JSONDecodeError as e: # Manejo elegante  
        print(f"Error: JSON inválido. {e}",  
              file=sys.stderr) sys.exit(1)
```



## Parte 4: Output & Serialización

Cómo devolver los datos procesados al Low-code.  
La regla sagrada del STDOUT.

**CRÍTICO**

## La Regla de Oro del Output

### SOLO UN PRINT

El script debe imprimir en consola (STDOUT) **única y exclusivamente** el JSON final.

**✗ Mal**

```
print("Conectando ... ")
print("Procesando ... ")
print(json_result)
```

El Low-code recibirá: "Conectando...Procesando...{...}" y fallará al parsear.

**✓ Bien**

```
# Logs a stderr o archivo
print(json.dumps(result))
```

Salida limpia JSON válido.

Serialización

# json.dumps()

Convierte tu diccionario/lista de Python en un String JSON válido.

## Tip Pro: default=str

Por defecto, dumps falla con objetos `datetime` o `decimal`. Usa un conversor `default=str` para transformarlos a texto automáticamente.

```
import json from datetime import datetime data = { "now": datetime.now(), "price": 10.5 } # default=str convierte fechas a string ISO json_output = json.dumps(data, default=str) print(json_output) # {"now": "2023-10-25 10:00... ", "price": 10.5}
```

**Sistema**

# Códigos de Salida (Exit Codes)

¿Cómo sabe Power Automate si el script funcionó o falló? No lee el texto, lee el **Exit Code**.

## 0 Éxito (Success)

El script terminó bien. El flujo continúa.

## 1 Error (Failure)

Algo falló. El flujo se detiene o va a la rama de error.

```
import sys
if todo_bien: print(json_resultado)
sys.exit(0) # Opcional (defecto)
else:
    sys.stderr.write("Error grave de conexión")
    sys.exit(1) # Fuerza fallo en Low-code
```

**Logging**

# STDERR: El Canal de Logs

Si necesitas imprimir logs ("Paso 1 completado", "Advertencia...") sin romper el JSON de salida, usa **Standard Error (stderr)**.

CANAL 1 (STDOUT) → Payload      CANAL 2 (STDERR) → Logs/Consola

```
import sys  
print("Esto va al JSON", file=sys.stdout)  
print("Esto es un log seguro", file=sys.stderr)
```

La mayoría de herramientas Low-code capturan STDERR y lo muestran en los logs de ejecución, pero no lo meten en la variable de respuesta.

**Resumen**

# Cheat Sheet del Protocolo

Acción	Código Python	Nota
Leer Input	<code>json.loads(sys.stdin.read())</code>	Mejor que sys.argv
Enviar Output	<code>print(json.dumps(data))</code>	Único print del script
Loguear	<code>print("Log", file=sys.stderr)</code>	No rompe el JSON
Reportar Fallo	<code>sys.exit(1)</code>	El Low-code sabrá que falló

**Siguiente Módulo: Casos Prácticos →**



CASO PRÁCTICO A

# Procesamiento Inteligente de Documentos (IDP)

Automatizando la extracción de datos de CVs en PDF.

**De Outlook a Excel sin tocar un teclado.**

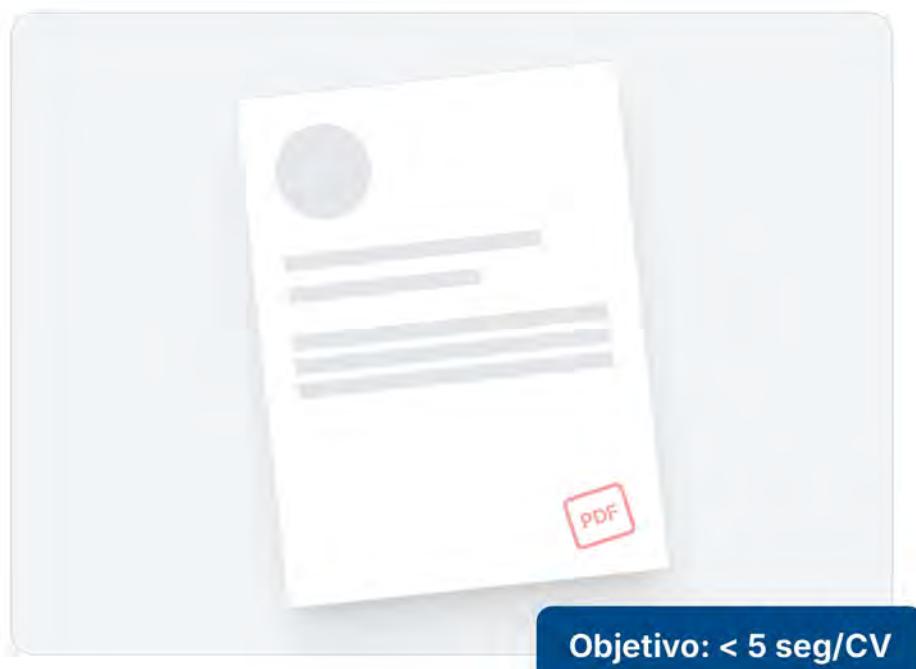
**Problema**

## La Pesadilla de RRHH

Cada día llegan 50+ correos con CVs adjuntos.  
Alguien tiene que abrir cada PDF, copiar el email y  
el teléfono, y pegarlo en un Excel.

**Por qué falla el Low-code puro:**

- ✗ "Get Text from PDF" es caro (AI Builder: \$500/mes).
- ✗ El texto viene sucio ("Email: juan@..." vs "juan@...").
- ✗ Regex en Power Automate es una tortura.

**Objetivo: < 5 seg/CV**

**Plan**

# Flujo de Trabajo Híbrido

**1. Outlook**

Nuevo Email

**2. Save File**

Guardar Temporal

**3. Python Script**

Extraer Texto + Regex

**4. Excel**

Add Row

Setup

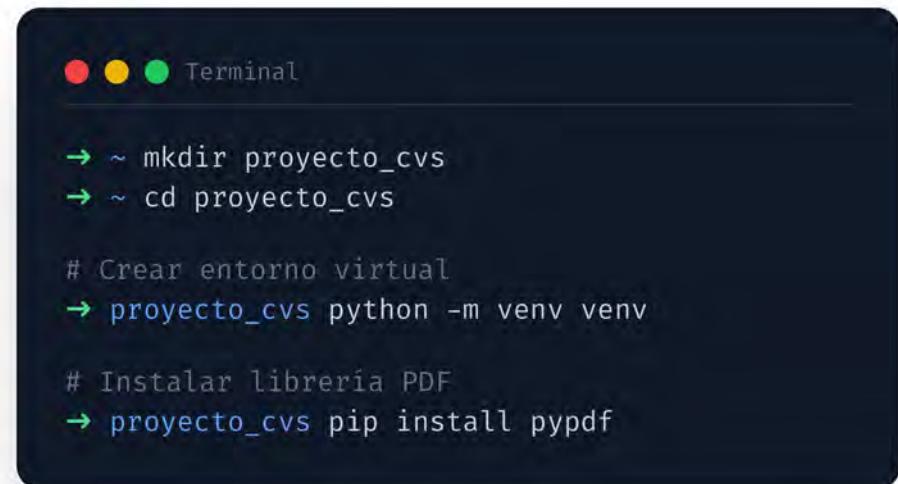
# Preparando la Cocina

## Ingredientes Necesarios

Antes de abrir Power Automate, necesitamos tener Python listo con las librerías adecuadas para leer PDFs.

### Nota Importante:

Usaremos pypdf (moderno) en lugar de 'PyPDF2' (obsoleto), aunque la sintaxis es similar.



```
Terminal

→ ~ mkdir proyecto_cvs
→ ~ cd proyecto_cvs

# Crear entorno virtual
→ proyecto_cvs python -m venv venv

# Instalar libreria PDF
→ proyecto_cvs pip install pypdf
```

**Paso 1 & 2**

# Captura y Guardado

## Configuración PAD

**Action:** Retrieve email messages

Folder: Inbox | Save Attachments: True

**Action:** For each (Attachment)

Iteramos sobre cada archivo descargado.

**Variable Clave:** %CurrentAttachment%

Esta variable contiene la RUTA completa del archivo en disco (ej. C:\Tmp\cv\_juan.pdf).

## El Truco del Archivo Temporal

Python no puede leer el "archivo en la nube" de Outlook directamente.

**Primero debemos descargarlo a una carpeta temporal local.**  
Luego le pasamos la ruta a Python.



C:\Temp\file.pdf

**Paso 3**

# Extracción de Texto (OCR)

cv\_parser.py

```
import sys from pypdf import PdfReader # 1. Recibimos la ruta del
archivo desde PAD # PAD enviará: python cv_parser.py "C:
\Temp\cv.pdf" pdf_path = sys.argv[1] try: # 2. Abrimos el PDF
reader = PdfReader(pdf_path) # 3. Extraemos texto de todas las
páginas full_text = "" for page in reader.pages: full_text +=
page.extract_text() + "\n" # Debug (Solo stderr, nunca stdout!)
import sys print(f"Leídos {len(full_text)} caracteres",
file=sys.stderr) except Exception as e: sys.stderr.write(str(e))
sys.exit(1)
```

## Librería pypdf

Es ligera, puramente Python (no requiere instalar binarios como Poppler) y funciona muy bien para PDFs de texto nativo (Word → PDF).

## Limitación

Si el PDF es una **imagen escaneada** (foto), pypdf no extraerá nada. En ese caso necesitaríamos pytesseract (OCR real), que es más complejo de instalar.

Paso 4

## La Magia del Regex

### Encontrando la aguja en el pajar

El texto extraído es un bloque sucio sin estructura.

Usamos **Expresiones Regulares** para "pescar" los datos.

```
" ... experiencia en ventas. Contacto:  
juan.perez@gmail.com o llamar al +34 600 123 456 para  
entrevista ... "
```

```
import re # Patrón Email (Simplificado)  
email_regex = r'[\w\.-]+@[ \w\.-]+\.\w+' # Patrón  
Teléfono (Flexible) # Captura: +34 600 ... ó  
600-123 ... phone_regex = r'\+?(\d[\d\-. ]+)?  
(\([\d\-. ]+\))?\d\-. ]+\d' emails =  
re.findall(email_regex, full_text) phones =  
re.findall(phone_regex, full_text) # Tomamos el  
primero encontrado (Heurística simple)  
candidate_email = emails[0] if emails else "No  
encontrado" candidate_phone = phones[0] if  
phones else "No encontrado"
```

**Paso 5**

## Empaquetando la Respuesta

Finalmente, convertimos las variables de Python en un objeto JSON limpio que Power Automate pueda entender.

```
output_data = { "status":  
    "success", "data": { "email":  
        candidate_email, "phone":  
        candidate_phone, "text_len":  
        len(full_text) } } import json  
print(json.dumps(output_data))
```

**STDOUT**

```
{  
    "status": "success",  
    "data": {  
        "email": "juan.perez@gmail.com",  
        "phone": "+34 600 123 456",  
        "text_len": 1540  
    }  
}
```

**Paso 6**

## Integración: "Convertir JSON"

En Power Automate, la variable %PythonScriptOutput% es solo TEXTO. El robot no sabe que es un objeto con propiedades.

**Acción Requerida:**

Usar la acción "Convert JSON to custom object".

- Input: %PythonScriptOutput%
- Output: %JsonAsCustomObject%

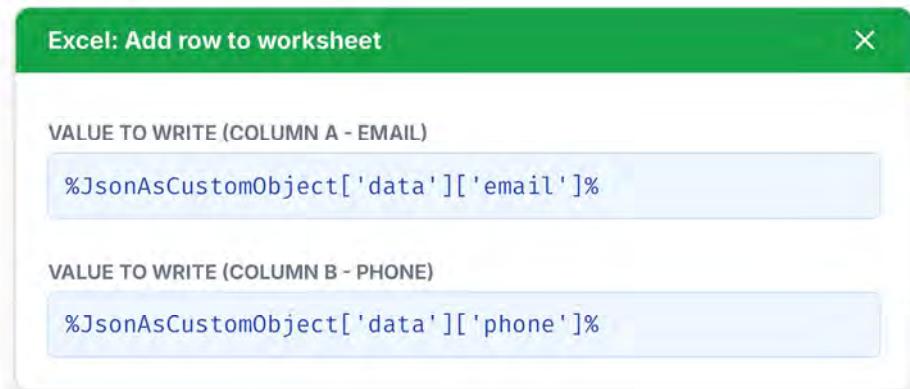


**Paso 7**

## Acción Final: Excel

### Cerrando el Ciclo

Ahora podemos usar las propiedades del objeto JSON directamente en la acción de Excel.



A (Email)	B (Phone)	C (Status)
ana@test.com	600111222	Procesado
juan.perez@...	+34 600...	...

# Live Demo Time

```
> Starting Flow 'CV_Processor_V1' ...
> Trigger: New Email received from 'Juan Perez'.
> Downloading attachment 'cv_juan.pdf' to C:\Temp\...
> Executing Python Script ...
  STDERR: Leidos 1540 caracteres.
  STDERR: Email found: juan.perez@gmail.com
> Python Script finished. Exit Code: 0.
> Parsing JSON response ...
> Excel Row Added successfully.
-
```

**Futuro**

# Mejorando el Robot (LLM)

## Del Regex a la IA

El Regex falla si el formato cambia (ej. "Tlf:" en lugar de "Teléfono:").

El siguiente paso lógico es sustituir el Regex por una llamada a **OpenAI** dentro del script.

**Prompt:**

*"Extrae el nombre, email y skills principales de este texto y devuélvelo en JSON: {full\_text}"*

**Reto para casa**

Modifica el script `script_cv_parser.py` para importar `openai` y usar GPT-3.5-turbo en lugar de `re`.

**Ver Solución en GitHub**



CASO PRÁCTICO B

# Enriquecimiento de Datos & Lead Scoring

Conectando APIs externas y aplicando lógica de negocio compleja.

**De Typeform a Slack con Cerebro Python.**

**Escenario**

# ¿Es este cliente potencial valioso?

Un usuario rellena un formulario simple (solo email). Queremos saber **automáticamente** si vale la pena que un vendedor lo llame.

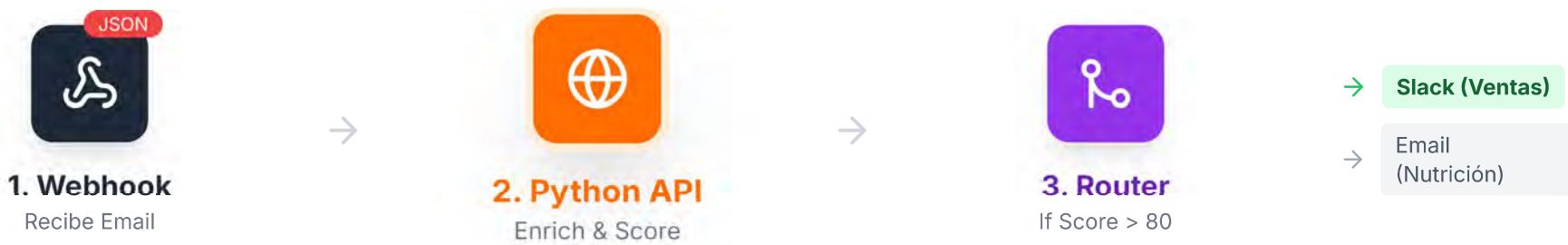
## Reglas de Negocio (Scoring):

- Base: 10 puntos.
- Si es sector "Tecnología": +50 puntos.
- Si tiene > 100 empleados: +30 puntos.
- Si usa correo gratuito (@gmail): -90 puntos.



**Diseño**

# El Flujo de Datos



**Herramientas**

# La Librería 'requests'

## HTTP para Humanos

En lugar de usar bloques visuales complejos para manejar cabeceras, timeouts y JSON parsing, usamos la librería estándar de facto en Python.

### ¿Por qué no usar el bloque HTTP de Make?

Porque necesitamos aplicar lógica (Scoring) **sobre** la respuesta de la API antes de devolverla. Si lo hacemos en Make, gastamos operaciones extra.

```
import requests # Low-code visual vs Python
response = requests.get( "https://
api.clearbit.com/v2/companies/find",
params={"domain": "tesla.com"},
headers={"Authorization": "Bearer sk_..."} )
data = response.json() # Listo. Ya tenemos un
diccionario Python.
```

**Paso 1**

# Recibiendo el Lead

## Configuración del Webhook

Configuramos el Webhook en nuestra herramienta (n8n/Make). Typeform enviará esto:

```
{ "event_id": "01J...", "form_response": {  
  "answers": [ { "type": "email", "email":  
    "juan@tesla.com" } ] }
```

## Python Input Parsing

```
import sys, json # 1. Leemos el Webhook completo  
raw_input = sys.stdin.read() webhook_data =  
json.loads(raw_input) # 2. Extraemos el email  
(Navegación segura) email =  
webhook_data.get('form_response', {}) \ .get('answers',  
[{}])[0] \ .get('email') if not email: sys.exit(1) #  
Abortar si no hay email
```

**Paso 2**

## Enriquecimiento (La Llamada)

### Obteniendo datos de la nada

Usamos el dominio del email (tesla.com) para preguntar a una API de enriquecimiento (ej. Clearbit, Apollo, o un Mock gratuito) quién es esta empresa.

**Tip de Seguridad:** La API Key no se escribe en el código. Se lee de `os.environ`.

```
import os
api_key = os.environ.get('CLEARBIT_KEY')
domain = email.split('@')[1] # Validar correos gratuitos
free_providers = ['gmail.com', 'hotmail.com']
if domain in free_providers: company_data = None
else: # Llamada Real
    resp = requests.get(
        f"https://company.stream/api/v2/companies",
        params={"domain": domain}, headers={"Auth": api_key})
    company_data = resp.json()
```

**Paso 3**

# Algoritmo de Scoring

## ¿Por qué Python aquí?

Implementar esta lógica en Low-code requeriría un árbol de decisiones visual ("Switch" o "If/Else") gigante y difícil de mantener. En Python son 5 líneas claras.

Sector IT	+50 pts
Empleados > 50	+30 pts
Email Gratis	-90 pts

```
score = 10 # Base if not company_data: score -= 90 # Castigo por email personal sector = "Unknown" else: sector = company_data.get('category', '').get('sector') employees = company_data.get('metrics', {}).get('employees', 0) if "Technology" in sector: score += 50 if employees > 50: score += 30 # Normalizar (Max 100, Min 0) score = max(0, min(100, score))
```

**Paso 4**

## La Respuesta JSON

Devolvemos no solo el Score, sino también una recomendación de acción ("next\_step") para facilitar la vida al Router del Low-code.

STDOUT

```
{  
    "lead_email": "juan@tesla.com",  
    "enrichment": {  
        "company": "Tesla Inc.",  
        "sector": "Automotive/Tech"  
    },  
    "scoring": {  
        "total": 90,  
        "tier": "A"  
    },  
    "next_step": "SALES_ALERT"  
}
```

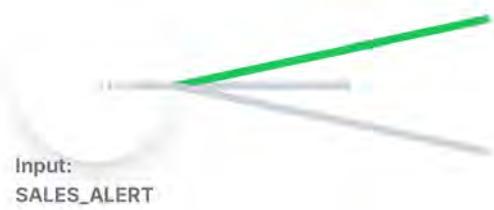
**Paso 5**

## Orquestación Visual

### El Router Inteligente

Ahora que Python hizo el trabajo sucio, el Low-code solo tiene que leer el campo `next_step` y dirigir el tráfico.

**Lógica Simple:**  
`Switch (json.next_step)`



**Resiliencia**

# ¿Qué pasa si la API falla?

## Escenario de Fallo

La API de Clearbit está caída (Error 500) o nos quedamos sin créditos (Error 402).

**No queremos que el flujo se rompa por completo.**

## Solución Defensiva:

```
try: resp = requests.get( ... )
resp.raise_for_status() # Lanza error si no es
200 company_data = resp.json() except
requests.exceptions.RequestException: # Fallback
suave (Degradación grácil) # Asumimos que no
sabemos nada, score base. print(f"WARNING: API
caída", file=sys.stderr) company_data = {} #
Diccionario vacío # El script NO muere (exit 0),
el flujo continúa # pero el score será bajo
(default).
```

# El Resultado Final



[Cierre](#)

# Lo que hemos aprendido

## 1. Request > HTTP Module

Usar Python para llamadas API nos permite limpiar, validar y enriquecer los datos **antes** de devolverlos al flujo visual.

## 3. Degradación Grácil

Manejar errores con 'try/except' evita que todo el proceso se detenga por un fallo de red externo.

## 2. Scoring Determinista

Centralizar la lógica de negocio (puntos) en un script hace que sea fácil de ajustar mañana sin mover 20 nodos visuales.

## Siguiente Paso

¿Cómo lo subimos a producción?



MÓDULO 6

# Buenas Prácticas & Gobernanza

Cómo evitar que tu automatización explote en producción.  
Seguridad, Mantenimiento y "Future-Proofing".

**Error #1**

## "En mi máquina funciona"

### Dependency Hell

Desarrollas tu script hoy usando pandas 2.0.

Dentro de 6 meses, actualizas Python globalmente y pandas cambia a 3.0.

**¡BOOM! ⚡ Tu automatización crítica se rompe un martes a las 3 AM.**



pip install pandas (Global)



```
python -m venv venv  
source venv/bin/activate  
pip install -r requirements.txt
```

**Solución**

# Congelar el Tiempo (Freezing)

Cada script debe viajar con su "pasaporte" de librerías. El archivo requirements.txt asegura que el servidor de producción tenga **exactamente** las mismas versiones que tu PC.

**Tip Pro:**

No pongas solo pandas. Pon pandas=2.0.3. Eso es determinismo.

requirements.txt

```
pandas=2.0.3
requests=2.31.0
openpyxl=3.1.2
pypdf=3.15.1
# Comentario: Necesario para reportes
jinja2>=3.1
```

**Seguridad**

# El Pecado Capital: Hardcoding

**Nunca hagas esto:**

```
api_key = "sk-1234567890abcdef" # ← PELIGRO password = "admin123" response = requests.get(url,  
auth=(user, password))
```

INSEGURERO

Si compartes este script, subes una captura a Slack o lo guardas en Git, **te han hackeado**.

Seguridad

# La Solución: Variables de Entorno

## Separa el Código del Secreto

El código define **qué** hacer. El entorno define **con qué credenciales** hacerlo.

- 💻 En PAD:Usa variables "Sensitive" (encriptadas).
- ☁️ En Make/n8n:Usa "Credentials" o "Environment Variables".

```
import os # Si la variable no existe, el
          script falla seguro api_key =
os.environ.get('STRIPE_KEY') if not api_key:
    raise ValueError("Falta STRIPE_KEY")
```

Safe to Share on Git

Arquitectura

# El Concepto Sagrado: Idempotencia

*"Si ejecuto el script 2 veces, no debe cobrar al cliente 2 veces."*

## Código Peligroso

```
# Siempre envía email  
send_email("Factura", cliente)
```

Si el script falla en la línea siguiente y reintentas, el cliente recibe 2 emails.

## Código Idempotente

```
if not cliente.ya_recibio_email:  
    send_email("Factura", cliente)  
    marcar_como_enviado(cliente)
```

Puedes reintentar 100 veces, solo envía 1 email.

**Mantenimiento**

# Logging: La Caja Negra

## ¿Por qué falló ayer?

Sin logs, eres ciego. Pero cuidado:  
**STDOUT** es para datos (JSON).  
**STDERR** es para logs humanos.

```
[INFO] Iniciando proceso para ID 555 ...
[WARN] API tardó 2s (lento)
[ERROR] File Not Found
```

```
import sys
def log(msg, level="INFO"):
    # Escribir a STDERR para no romper el JSON
    print(f"[{level}] {msg}", file=sys.stderr)
    log("Conectando a DB ...") try: db.connect()
    except: log("Fallo DB", "CRITICAL") sys.exit(1)
```

**Pro**

## Trata la Automatización como Software



### No guardes código en la herramienta visual

Si escribes 500 líneas de Python dentro de la cajita de texto de n8n/PAD, estás creando deuda técnica. ¿Cómo vuelves a la versión de ayer si rompes algo?

#### El Flujo Correcto

1. Escribir en VS Code.
2. Commit & Push a GitHub.
3. n8n/PAD hace "Pull" o lee el archivo.

#### El Flujo Amateur

Copy-Paste directo en el navegador.

Cultura

# Documentación para el "Tú del Futuro"

Dentro de 3 meses, no recordarás por qué pusiste ese `sleep(5)` ahí.

## Docstrings & Type Hints

Python permite documentar qué entra y qué sale. Úsalo.

```
def calcular_impuesto(monto: float) → float:  
    """ Calcula el IVA (21%). Args: monto (float):  
    Precio base sin impuestos. Returns: float:  
    Precio final con impuestos. """ # FIX: Añadimos  
    0.01 por error de redondeo return (monto * 1.21)  
    + 0.01
```

**Recap**

# Lo que hemos construido hoy



## El Puente

Aprendimos a conectar Low-code (Trigger/Action) con Python (Processing) usando JSON.



## La Potencia

Vimos cómo Pandas, Regex y Requests superan cualquier limitación visual.



## La Robustez

Entendimos la importancia de los entornos, secretos e idempotencia.

Resources

# Tu Kit de Supervivencia

## Lectura Obligatoria

[Pandas User Guide](#) - Biblia de datos.

[Real Python](#) - Tutoriales excelentes.

[n8n Community](#) - Recetas de workflows.

## Entregables de Hoy

 [cv\\_parser.py](#)

 [lead\\_score.py](#)

 [n8n\\_workflow\\_template.json](#)



## Q & A

¿Dudas sobre vuestro caso de uso específico?  
¿Miedo a romper producción?

@TuUsuario

[github.com/tu-repo](https://github.com/tu-repo)