# 🤖 Servicio Lanzador - Proyecto SAM

## Visión General

El **Servicio Lanzador** es el componente central de ejecución y sincronización del ecosistema SAM (Sistema Automático de Robots). Opera como un servicio de Windows persistente, asíncrono y de alto rendimiento, diseñado para orquestar la ejecución de robots RPA en un entorno de Automation Anywhere 360 (AA360).

Sus responsabilidades se dividen en tres ciclos operativos independientes y concurrentes, cada uno configurable en su intervalo de ejecución:

1. **Ciclo de Sincronización de Tablas Maestras**: Mantiene la base de datos de SAM actualizada con la realidad del Control Room de AA360. Descubre nuevos robots y equipos (VMs), y actualiza sus estados y licencias.
2. **Ciclo de Lanzamiento de Robots**: Es el núcleo del servicio. Identifica los robots que deben ejecutarse y los despliega de forma **concurrente y paralela**, maximizando la eficiencia. Además, implementa una **lógica de reintentos inteligente** para superar fallos transitorios, como un dispositivo temporalmente inactivo.
3. **Ciclo de Conciliación de Estados**: Actúa como un mecanismo de auditoría y autocorrección. Revisa periódicamente las ejecuciones que SAM considera "activas" y verifica su estado real en AA360, actualizando la base de datos para corregir cualquier discrepancia.

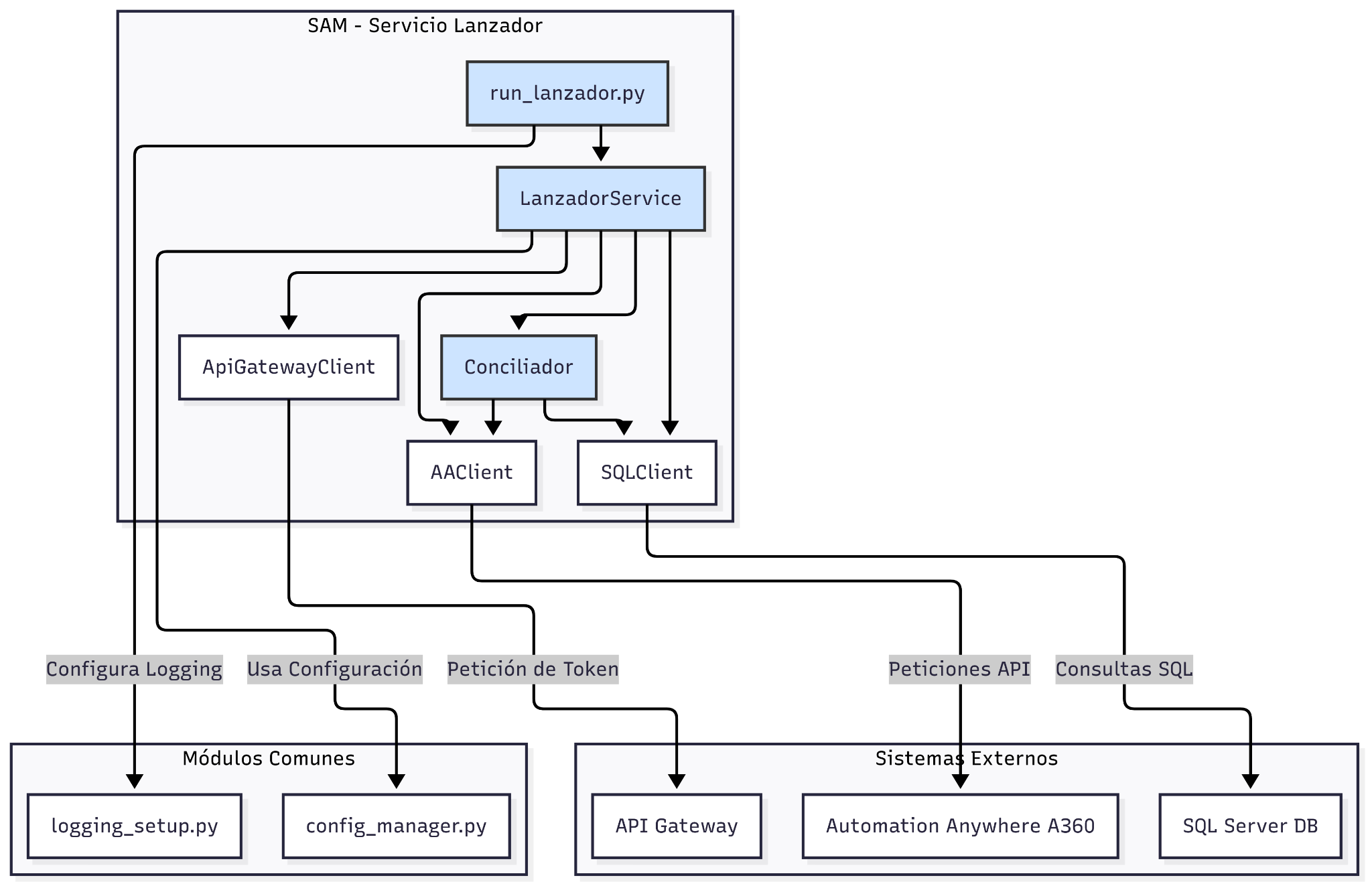
Una característica clave es la integración con un **API Gateway** externo para gestionar la seguridad de los callbacks. Antes de lanzar un robot, el servicio obtiene un token de autenticación (OAuth 2.0) que se inyecta en la petición de despliegue. Esto asegura que cuando AA360 notifique la finalización de un robot, la llamada a nuestro servicio de Callback sea segura y autenticada.

## Arquitectura y Componentes

El servicio está construido sobre una arquitectura modular y asíncrona utilizando asyncio de Python, lo que le permite realizar múltiples tareas de red (llamadas a APIs, consultas a BD) de manera eficiente y sin bloqueos.

### Diagrama de Componentes

Este diagrama ilustra las principales clases y módulos del servicio y cómo interactúan entre sí y con sistemas externos.



* **run\_lanzador.py**: Punto de entrada del servicio. Responsable de inicializar la configuración, el logging y el manejador de señales para un cierre ordenado. Lanza la instancia de LanzadorService.
* **LanzadorService (main.py)**: Orquestador principal. Gestiona los tres ciclos de vida (sincronización, lanzamiento, conciliación) como tareas asíncronas.
* **AAClient (aa\_client.py)**: Cliente HTTP asíncrono para interactuar con la API de AA360. Encapsula la lógica de autenticación (token de AA360), paginación y los endpoints para desplegar bots, obtener listas de robots, equipos y usuarios.
* **ApiGatewayClient (api\_gateway\_client.py)**: Cliente HTTP asíncrono dedicado a la comunicación con el API Gateway. Su única responsabilidad es obtener y gestionar el ciclo de vida del token de autorización (OAuth 2.0) necesario para los callbacks.
* **SQLClient (sql\_client.py)**: Conector a la base de datos SQL Server. Proporciona métodos para ejecutar consultas y Stored Procedures, con lógica de reintentos y gestión de conexiones segura para entornos multihilo/asíncronos.
* **Conciliador (conciliador.py)**: Implementa la lógica de conciliación. Utiliza el AAClient y SQLClient para comparar los estados de las ejecuciones y corregir inconsistencias.
* **Módulos Comunes (utils/)**: Incluyen ConfigManager para cargar variables de entorno de forma jerárquizada y logging\_setup para una configuración de logs estandarizada y robusta.

## 

## Flujo de Ejecución

El siguiente diagrama de secuencia detalla el flujo de operaciones del ciclo más importante: el **Ciclo de Lanzamiento**, destacando su naturaleza paralela y su capacidad de reintentos.

### Diagrama de Secuencia: Ciclo de Lanzamiento Concurrente y Robusto

**Pasos del Flujo:**

1. El LanzadorService inicia su ciclo de lanzamiento.
2. Consulta a la base de datos para obtener la lista de robots que deben ejecutarse.
3. Si hay robots, solicita las cabeceras de autorización al ApiGatewayClient.
4. El servicio crea una tarea de despliegue para **cada robot** en la lista.
5. Todas estas tareas se ejecutan **en paralelo** (hasta un límite configurable) usando asyncio.gather.
6. Para cada tarea de despliegue:
   * Se llama a la API de AA360.
   * Si la API devuelve un error específico (ej. HTTP 400 - Device not active), la tarea espera un tiempo configurable y **reintenta la llamada**.
   * Si el despliegue es exitoso, se registra el deploymentId en la base de datos.
   * Si el despliegue falla definitivamente (tras agotar los reintentos), se registra el error.
7. El ciclo espera a que todas las tareas paralelas terminen antes de concluir.

## Detalle Técnico

* **Concurrencia y Asincronía**: El servicio aprovecha asyncio para lanzar múltiples peticiones de despliegue a la API de AA360 de forma simultánea. Esto se gestiona a través de asyncio.gather, que procesa las tareas en lotes, respetando un límite de concurrencia (LANZADOR\_MAX\_CONCURRENTES\_DEPLOYMENTS) para no sobrecargar el sistema de destino.
* **Resiliencia y Reintentos**:
  + **Reintentos de Despliegue**: El servicio es capaz de reintentar un despliegue que falle por errores transitorios específicos, como un HTTP 400 indicando que el dispositivo de destino no está activo. El número de reintentos y el tiempo de espera son configurables, lo que permite al sistema recuperarse de situaciones temporales (ej. un reinicio de VM) sin intervención manual.
  + **Base de Datos**: El SQLClient puede reintentar automáticamente las consultas que fallan por errores transitorios comunes como *deadlocks*.
  + **API Gateway**: Si la obtención del token falla, el servicio no se detiene. Lanza los robots sin las cabeceras de autorización para el callback, confiando en el **Ciclo de Conciliación** como mecanismo de respaldo.
* **Gestión de Tokens (AA360 y API Gateway)**: Ambos clientes (AAClient y ApiGatewayClient) gestionan sus tokens de forma "lazy" (perezosa). Solo solicitan un token cuando es estrictamente necesario y utilizan un asyncio.Lock para evitar *race conditions*.
* **Configuración Centralizada**: Toda la configuración se gestiona a través de ConfigManager y se carga desde archivos .env, permitiendo modificar parámetros clave sin alterar el código.

## Configuración (.env)

Para que el servicio Lanzador funcione correctamente, las siguientes variables de entorno deben estar definidas:

# --- Configuración de Logging ---

LOG\_DIRECTORY=C:/RPA/Logs/SAM

LOG\_LEVEL=INFO

APP\_LOG\_FILENAME\_LANZADOR=sam\_lanzador\_app.log

# --- Base de Datos SAM (SQL Server) ---

SQL\_SAM\_HOST=TU\_SERVIDOR\_SQL

SQL\_SAM\_DB\_NAME=SAM

SQL\_SAM\_USER=tu\_usuario

SQL\_SAM\_PASSWORD=tu\_contraseña

# --- Automation Anywhere A360 ---

AA\_URL=https://tu-instancia.my.automationanywhere.digital

AA\_USER=sam\_user

AA\_PWD=sam\_password

AA\_URL\_CALLBACK=http://tu\_ip\_publica:8008/sam\_callback

# --- API Gateway para Callbacks ---

API\_GATEWAY\_URL=https://apiinternos.movistar.com.ar/telefonica/api/v1/oauth2/token

API\_GATEWAY\_CLIENT\_ID=...

API\_GATEWAY\_CLIENT\_SECRET=...

API\_GATEWAY\_SCOPE=scope1

# --- Parámetros del Lanzador ---

LANZADOR\_INTERVALO\_LANZADOR\_SEG=30

LANZADOR\_INTERVALO\_CONCILIADOR\_SEG=180

LANZADOR\_INTERVALO\_SYNC\_TABLAS\_SEG=3600

LANZADOR\_HABILITAR\_SYNC=True

# Parámetros de rendimiento y resiliencia

LANZADOR\_MAX\_CONCURRENTES\_DEPLOYMENTS=10

LANZADOR\_MAX\_REINTENTOS\_DEPLOY=1

LANZADOR\_DELAY\_REINTENTO\_DEPLOY\_SEG=15

## Instalación y Despliegue

1. **Prerrequisitos**: Python 3.8+.
2. **Entorno Virtual**: Se recomienda crear y activar un entorno virtual.
3. **Instalar Dependencias**: pip install -r requirements.txt.
4. **Configurar .env**: Crear y configurar el archivo .env en la raíz del proyecto.
5. **Ejecutar el Servicio**: python src/lanzador/run\_lanzador.py.
6. **Despliegue en Producción**: Se recomienda usar **NSSM** para instalarlo como un servicio de Windows.